Interfaces, Injeção de dependência e Container IoC: Use Spring4D e reduza o acoplamento

Clube Delphi

Ano 13 :: Edição 154 :: R$ 14,90

DEVMEDIA

Introdução ao UniDAC
Explore mais uma opção para acesso a dados

Não conhece o mORMt?
Construa um servidor RESTful em segundos!

Crie e consuma relatórios pela Web
DataSnap como um serviço

Flexibilidade com Design Patterns

Aplicando os padrões comportamentais

ISSN 1517990-7

DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DEVMEDIA

Clube Delphi

Ano 13 - 154ª Edição 2013 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Wesley Yamazack (yamazack@gmail.com)

Capa

Romulo Araujo

Diagramação

Janete Feitosa

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Artigo do tipo Mentoring

04 – Aplicação prática de Design Patterns
[*Guinther Pauli*]

Artigo no estilo Solução Completa

16 – Criação de relatórios via webservices e DataSnap
[*Leandra Mendes do Vale*]

Artigo no estilo Curso

24 – Introdução ao UniDAC – Parte 1
[*Fabrício Hissao Kawata*]

Artigo no estilo Solução Completa

32 – Exposição de objetos de negócio em JSON por servidor REST
[*Roberto Schneiders*]

Conteúdo sobre Boas Práticas

44 – Boas práticas para redução de acoplamento
[*Paulo Quicoli*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Aplicação prática de Design Patterns

Crie sistemas flexíveis com padrões comportamentais– Chain of Responsibility, Command, Iterator, Mediator e Memento

A intenção deste padrão é evitar o acoplamento do remetente de uma solicitação ao seu recepetor, ao dar a mais de um objeto a oportunidade de tratar essa solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate. Nesse padrão cada objeto receptor possui uma lógica descrevendo os tipos de solicitação que é capaz de processar e como passar adiante aquelas que requeiram processamento por outros receptores. A delegação das solicitações pode formar uma árvore de recursão, com um mecanismo especial para inserção de novos receptores no final da cadeia existente. Dessa forma, fornece um acoplamento mais fraco por evitar a associação explícita do remetente de uma solicitação ao seu receptor e dar a mais de um objeto a oportunidade de tratar a solicitação. Um exemplo da aplicação desse padrão é o mecanismo de herança nas linguagens orientadas a objeto: um método chamado em um objeto é buscado na classe que implementa o objeto e, se não encontrado, na superclasse dessa classe, de maneira recursiva. Como benefício do uso do padrão, ele permite determinar quem será o objeto que irá tratar a requisição durante a execução. Cada objeto pode tratar ou passar a mensagem para o próximo na cadeia. Dessa forma, o acoplamento é reduzido, dando ainda flexibilidade adicional na atribuição de responsabilidades a objetos.

Participantes:

• Handler – define uma interface para tratar solicitações;

• ConcreteHandler - trata de solicitações pelas quais é responsável; pode acessar seu sucessor;

• Cliente – inicia a solicitação para um objeto ConcreteHandler da cadeia;

Resumo DevMan

Porque esse artigo é útil:

Padrões de projeto comportamentais são extremamente úteis para criar softwares mais fáceis de serem mantidos, evoluídos, promovendo reutilização de código, uso de boas práticas de programação orientada a objetos, redução de custos com manutenção, já que permitem que alterações em um determinado sistema de software não quebrem outras funcionalidades, reduzindo a dependência entre classes, através do uso exaustivo de abstrações. Dessa forma, são úteis para serem aplicados em projetos de média e larga escala. Este artigo abordará os padrões de projeto comportamentais: Chain of Responsibility, Command, Iterator, Mediator e Memento. Estes padrões ajudam a tornar sistemas de software mais fáceis de serem mantidos e evoluídos. Se uma aplicação precisa ser modificada em um determinado momento, seja para manutenção corretiva ou evolutiva para atender novas funcionalidades, este processo precisa ser feito de forma disciplinada, correta, usando padrões, já que o custo da manutenção representa um porcentual elevado comparado ao custo total do sistema. Com isso, o uso de padrões de projeto, ou Design Patterns, torna um software mais fácil de ser mantido em longo prazo, reduzindo custos ou otimizando o retorno do investimento, apesar de exigir um maior esforço em um primeiro momento, durante o design do projeto a ser realizado. Este artigo apresenta os padrões de projeto das classes comportamentais e implementações de exemplos práticos reais em Delphi. Serão tratados os padrões GoF, destinados a resolver problemas com operações, comunicação e o relacionamento (no sentido de chamar métodos e passar dados) entre objetos complexos, que comprometem a reutilização de módulos do sistema e aumentam o acoplamento. No final do artigo, apresento ainda mais um padrão estrutural, o Proxy, de forma a complementar os padrões vistos na edição anterior (Adapter, Bridge, Composite, Decorator e Facade).

Vamos a um exemplo prático no Delphi. A **Listagem 1** mostra um framework de classes que representa as classes participantes do padrão. É um pequeno sistema de autenticação de usuários, sendo que temos três classes que realizam esse processo de autenticação: Formulário, Server e BancoDados. De fato, em um sistema real, normalmente realizamos a operação de autenticação em várias camadas do nosso sistema. Primeiramente, no formulário de login, onde fazemos uma verificação (validação) básica, por exemplo, se o usuário não deixou os campos necessários para a autenticação em branco, ou se os formatos correspondem a uma máscara ou expressão regular (que pode ser um JavaScript em uma aplicação Web por exemplo), enfim, uma validação simples. No servidor de aplicação, podemos ter uma validação a nível de código Delphi, mais complexa, que possa validar por exemplo várias tentativas de login vindas de um mesmo cliente. Ou ainda, processar validações mais complexas. E finalmente, a checagem final que é feita no banco de dados, autenticando o usuário de acordo com a existência ou não das credenciais em uma determinada tabela. Note que cada camada da solução se encarrega de fazer um tratamento para as informações de usuário e senha, feito isso, repassa (ou não) esse tratamento para um sucessor na cadeia de responsabilidade, até que seja totalmente tratada ou que a cadeia seja finalizada (não exista mais um sucessor). No exemplo, estamos fazendo um diferente tratamento em cada classe (para fins didáticos é um tratamento simples), de forma que encadeamos as chamadas entre formulário, server e banco de dados, criando uma cadeia. O segredo do padrão fica por conta do participante Handler (manipulador), que é uma abstração para um participante da cadeia. Um objeto privado interno chamado _sucessor aponta para o objeto seguinte na cadeia, sendo este configurado através do método setSucessor. Então, o método Autenticar de cada Handler (formulário, server e banco de dados) precisa, após fazer sua própria autenticação, decidir se passará o processo para o membro seguinte na cadeia, testando o objeto _sucessor. Como ele é abstrato e possui um método virtual abstrato que é implementado por todos os membros (handlers), o polimorfismo se encarrega de chamar o método concreto adequado em tempo de execução. Por exemplo, se a classe server chamar o método Autenticar de _sucessor (do tipo base Autenticacao), será acionado o método polimórfico Autenticar

Listagem 1. Chain of Responsibility

```
unit uFramework;

interface

uses
  System.SysUtils;

type

 // Handler
 Autenticacao = class abstract
   protected _sucessor: Autenticacao;
   public procedure setSucessor(Sucessor: Autenticacao);
   public function Autenticar(UserName, Password: string): boolean;
   virtual; abstract;
 end;

 // ConcreteHandler
 Formulario = class(Autenticacao)
   public function Autenticar(Usuario, Senha: string): boolean; override;
 end;

 // ConcreteHandler
 Server = class(Autenticacao)
   public function Autenticar(Usuario, Senha: string): boolean; override;
 end;

 // ConcreteHandler
 BancoDados = class(Autenticacao)
   public function Autenticar(Usuario, Senha: string): boolean; override;
 end;

implementation

{ Autenticacao }

procedure Autenticacao.setSucessor(Sucessor: Autenticacao);
begin
  self._sucessor := Sucessor;
end;

{ Formulario }

function Formulario.Autenticar(Usuario, Senha: string): boolean;
begin
  result := (Usuario = 'guinther') and (Senha = '123');
  WriteLn('Validação feita no Formulário:' + BoolToStr(result,true));
  // encadeia chamada
  if (_sucessor <> nil) then
    result := _sucessor.Autenticar(Usuario, Senha);
end;

{ Server }

function Server.Autenticar(Usuario, Senha: string): boolean;
begin
  result := (Usuario = 'guinther') and (Senha = '123');
  WriteLn('Validação feita no AppServer:' + BoolToStr(result,true));
  // encadeia chamada
  if (_sucessor <> nil) then
    result := _sucessor.Autenticar(Usuario, Senha);
end;

{ BancoDados }

function BancoDados.Autenticar(Usuario, Senha: string): boolean;
begin
  result := (Usuario = 'guinther') and (Senha = '123');
  WriteLn('Validação feita no Banco de Dados:' + BoolToStr(result,true));
  // encadeia chamada
  if (_sucessor <> nil) then
    result := _sucessor.Autenticar(Usuario, Senha);
end;

end.
```

de BancoDados, que é o objeto imediatamente seguinte na cadeia de responsabilidade. Note, mais uma vez, o uso do recurso de delegação da orientação a objetos, juntamente com abstração e polimorfismo, usados em boa parte dos padrões GoF.

A **Listagem 2** mostra como consumir este framework. Aqui simplesmente criamos cada um dos membros (Concrete Handlers), configuramos os seus sucessores, formando assim a cadeia, e chamamos o método virtual Autenticar do primeiro membro da fila (o formulário), que fará sua autenticação e repassará (ou não) a chamada (responsabilidade) para as demais classes encadeadas.

Como já feito para os outros padrões, vamos examinar onde o próprio Delphi usa esse padrão, ou a nível de VCL, IDE ou linguagem. Um exemplo claro: observe que o próprio tratamento estruturado de exceções do Delphi (try except) na verdade usa uma espécie de padrão bastante semelhante ao Chain of Responsibility. De fato, quando uma exceção é levantada em algum ponto da aplicação, ela vai descendo na Call Stack de métodos até que um o manipule sem levantá-lo novamente com raise. Forma-se assim uma cadeia de responsabilidade, cada método pode decidir por tratar aquele erro (uma responsabilidade) e decidir repassar ou não para o próximo na cadeia, no caso o sucessor é o método imediatamente abaixo que o invocou. A **Figura 1** ilustra o padrão Chain of Responsibility.

Listagem 2. Usando um Chain of Responsibility

```
program ChainOfResponsability;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  frm: Formulario;
  app: Server;
  BD: BancoDados;
  Usuario, Senha: string;
begin
  Writeln('Usuario:');
  Readln(Usuario);
  Writeln('Senha:');
  Readln(Senha);
  frm := Formulario.Create();
  app := Server.Create();
  BD := BancoDados.Create();
  frm.setSucessor(app);
  app.setSucessor(BD);
  // passando requisição para cadeia de responsabilidade
  frm.Autenticar(Usuario,Senha);
  Readln;
end.
```

Command

A intenção do padrão Command é encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas. A chave deste padrão é uma classe abstrata Command, a qual declara uma interface para execução de operações. Na sua forma mais simples, esta interface inclui uma operação abstrata Execute. As subclasses concretas de Command especificam um par receptor-ação através do armazenamento do receptor como uma variável de instância, e pela implementação de Execute para invocar a solicitação. O receptor tem o conhecimento necessário para poder executar a solicitação. Entre os benefícios do uso do padrão, ele desacopla o objeto que invoca a operação daquele que sabe como executá-la; além disso, é fácil criar novos comandos porque não é necessário mudar classes existentes.

Participantes:

- Command - declara uma interface para executar a operação;
- ConcreteCommand – define uma vinculação entre o objeto Receiver (receptor) e uma ação; Implementa Execute invocando a operação correspondente em Receiver;
- Client – cria um objeto ConcreteCommand e configura o seu receptor;
- Invoker – objeto invocador, solicita ao Command a execução da solicitação;
- Receiver – sabe como executar as operações associadas a uma solicitação; qualquer classe pode funcionar como um receiver;

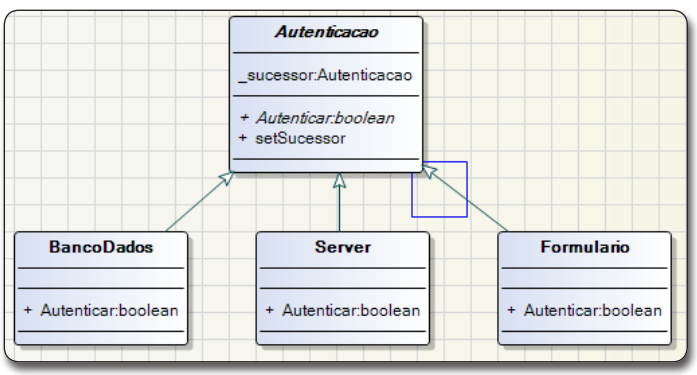

Figura 1. Estrutura de classes do exemplo usando Chain of Responsibility

A **Listagem 3** mostra o exemplo de um framework em Delphi que implementa o padrão Command. É um exemplo bastante semelhante ao anterior, da cadeia de responsabilidade, exceto pelo fato que aqui não estamos usando um encadeamento. Ao invés disso, temos um objeto Command que encapsula a própria ação de autenticação. Dessa forma, não é necessário que os participantes (aqui o Invoker e Receiver) herdem de uma classe base e por consequência implementem uma abstração polimórfica. O invocador é neste caso o formulário, que recebe um comando a ser executado. A seguir, ele envia esse comando ao recebedor, de forma totalmente desacoplada. Isto é, Invoker e Receiver não conhecem um ao outro, não possuem qualquer relação de herança, associação, composição ou mesmo alguma referência em algum parâmetro de algum método. Tudo o que estas classes conhecem é apenas e somente a classe Command (a abstração). Veja que com isso cada participante pode ter uma implementação diferente para

Listagem 3. Command

```
unit uFramework;

interface

type
 // Command
 Command = class abstract
   public procedure Execute(); virtual; abstract;
 end;

 // Invoker
 Formulario = class
   private _command: Command;
   public procedure setCommand(command: Command);
   public procedure ClickValidate();
 end;

 // Receiver
 Server = class
   public procedure ValidateUser();
 end;

 // ConcreteCommand
 ServerCommand = class(Command)
    private _server: Server;
    public constructor Create(server: Server);
    public procedure Execute(); override;
 end;

implementation

{ Formulario }

procedure Formulario.ClickValidate();
begin
  Writeln('Formulário - Iniciando validação do usuário');
  _command.Execute();
end;

procedure Formulario.setCommand(command: Command);
begin
  self._command := command;
end;

{ ServerCommand }

constructor ServerCommand.Create(server: Server);
begin
  self._server := server;
end;

procedure ServerCommand.Execute();
begin
  // delegando chamada a receiver
  _server.ValidateUser();
end;

{ Server }

procedure Server.ValidateUser();
begin
  Writeln('Servidor - Validando usuário');
end;

end.
```

uma validação, um formulário pode responder a essa responsabilidade no método que achar mais apropriado, diferentemente do exemplo anterior onde todos deveriam assinar (sobrescrever) o mesmo método virtual polimórfico. A **Listagem 4** mostra um exemplo de uso deste framework. Aqui criamos o comando a ser executado, quem será o invocador (formulário), o receptor (servidor) e disparamos uma ação no Invoker. O comando será então transferido, até chegar em seu receptor.

No Delphi existem muitos casos onde podemos achar implementações semelhantes a Command na VCL / FMX. Por exemplo, a comunicação entre objetos distribuídos em arquiteturas DataSnap utiliza comandos para desacoplar cliente e servidor. A biblioteca Indy utiliza comandos para envio de e-mails, por exemplo, desacoplando e tornando assíncrona a comunicação entre invocador e receptor. Aplicações Web utilizam o conceito de Response e Request, que são espécies de comandos enviados entre um browser e um servidor Web, seguindo uma especificação, também com o intuito de comunicar duas partes interessadas sem acoplá-las. A **Figura 2** ilustra os participantes do padrão Command.

Listagem 4. Usando um Command

```
program CommandApp;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  // receiver
  srv: Server;
  // invoker
  form: Formulario;
  // command
  cmd: Command;
begin
  // configura Receiver
  srv := Server.Create();
  // cria comando configurando o receiver
  cmd := ServerCommand.Create(srv);
  // configura o Invoker
  form := Formulario.Create();
  // configurar e executar o comando...
  form.setCommand(cmd);
  form.ClickValidate();
  ReadLn;
end.
```

Iterator

A intenção do padrão Iterator é fornecer uma maneira de acessar os elementos de um objeto agregado sequencialmente sem expor sua representação. Um objeto agregado, como uma lista, por exemplo, deveria sempre oferecer uma forma de acessar seus elementos

(dados) sem expor sua estrutura interna. Além disso, essa lista poderia ser examinada de diferentes formas, dependendo do que precisa ser realizado (procurar, ordenar etc.). Normalmente uma interface para cada operação sobrecarregaria a classe. Nesse caso um padrão uniforme (o Iterator) provê uma estratégia conhecida atualmente como "programação genérica", que visa separar explicitamente algoritmos de estruturas de dados. Os principais benefícios do padrão é promover a programação baseada em componentes, reutilização e aumento de produtividade.

Participantes:

• Iterator – define uma interface para acessar e percorrer elementos;

• ConcreteIterator – implementa a interface do Iterator; mantém o controle da posição corrente no percurso do agregado;

• Aggregate – define uma interface para criação de um objeto Iterator;

• ConcreteAggregate – implementa a interface de criação do Iterator para retornar uma instância do ConcreteCreator apropriado;

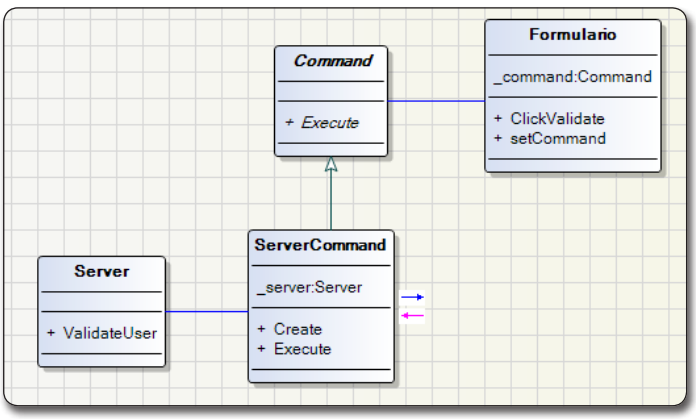

Figura 2. Estrutura de classes do exemplo usando Command

As **Listagens 5** e **6** trazem um exemplo de como implementar o padrão Iterator em Delphi. Aqui temos uma classe TCliente com dois atributos, ID e Nome. Temos então uma outra classe chamada TClienteList, que encapsula o acesso a uma coleção de objetos TCliente. Possui métodos para adicionar itens, remover e consultar objetos na coleção, que é privada e interna. Note que, por boas práticas, nunca expomos uma coleção em uma classe diretamente, mas fornecemos acesso a ela via métodos. Temos então uma classe chamada TClienteIterator, que permite varrer a classe TClienteList, listando todos os clientes nela armazenados. Para facilitar o exemplo, e reduzir o tamanho do código, implementamos aqui o padrão sem as abstrações, de forma que nem todos os participantes estão presentes. A **Figura 3** exibe o padrão Iterator.

Mediator

A intenção do padrão Mediator é definir um objeto que encapsula a interação entre um conjunto de objetos. Mediator promove o baixo acoplamento por manter objetos sem se referir um ao outro de forma explícita, e que lhe permite variar sua interação independentemente. O padrão Mediator pode ser usado quando: um conjunto de objetos se comunica de maneiras bem-definidas, porém complexas; as interdependências resultantes são desestruturadas e difíceis de entender. A reutilização de um objeto é difícil porque ele referencia e se comunica com muitos outros objetos; um comportamento que está distribuído entre várias classes deveria ser customizável, ou adaptável, sem excessiva especialização em subclasses. Benefícios: limita o uso de subclasses; desacopla colegas; simplifica o protocolo de objetos; abstrai a maneira como os objetos cooperam; ele centraliza o controle.

Participantes:

Mediator – define uma interface para comunicação com objetos Colleague;

• ConcreteMediator – implementa o comportamento cooperativo;

• Colleague classes – cada classe Colleague conhece seu objeto Mediator de outra forma; cada Colleague se comunica com seu Mediator sempre que, de outra forma, teria que se comunicar com outro colega;

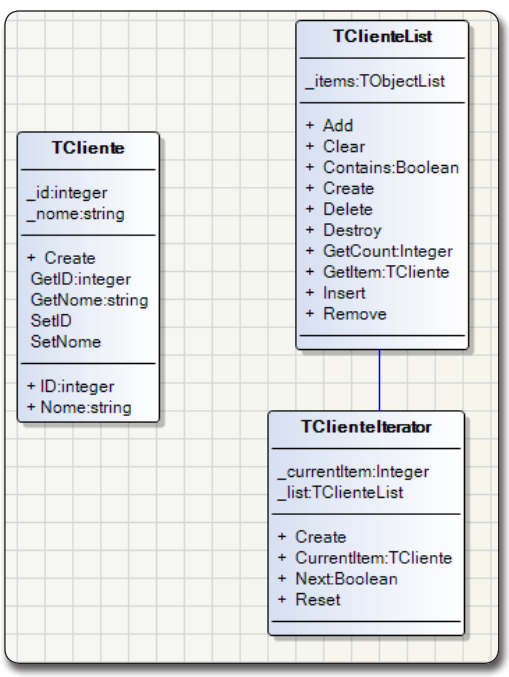

Figura 3. Estrutura de classes do exemplo usando Iterator

Para exemplificar o uso do Mediator vou utilizar um exemplo didático. Vamos imaginar na vida real duas classes que têm uma complexidade e atrito clássico na comunicação: usuários e suporte. De fato, ambos são objetos complexos que parecem falar línguas totalmente diferentes. Manter uma relação entre ambos é uma tarefa muito complexa, intermediar a comunicação entre ambos é papel para um mediador. O framework da **Listagem 7** mostra como ficariam estas classes implementadas através do padrão Mediator. Aqui temos as classes Colleague (colegas), que são as que precisam estabelecer a comunicação. Ambas as classes colegas, suporte e usuário, têm um método Notificar, que exibe

Listagem 5. Iterator

```
unit uFramework;
interface
uses
 System.Contnrs;
type

 TCliente = class
   private _id: integer;
   private _nome: string;
   private procedure SetID(const Value: integer);
   private procedure SetNome(const Value: string);
   private function GetID(): integer;
   private function GetNome(): string;
   public property ID: integer read GetID write SetID;
   public property Nome: string read GetNome write SetNome;
   public constructor Create(ID: integer; Nome: string);
 end;
 TClienteList = class
   private _items: TObjectList;
   public procedure Add(const Item: TCliente);
   public procedure Insert(Idx: Integer; const Item: TCliente);
   public procedure Delete(Idx: Integer);
   public procedure Remove(const Item: TCliente);
   public procedure Clear();
   public function Contains(const Item: TCliente): Boolean;
   public function GetCount(): Integer;
   public function GetItem(Idx: Integer): TCliente;
   public constructor Create();
   public destructor Destroy();
 end;
 TClienteIterator = class
   private _list: TClienteList;
   private _currentItem: Integer;
   public procedure Reset();
   public function Next(): Boolean;
   public function CurrentItem(): TCliente;
   public constructor Create(const List: TClienteList);
 end;
implementation
{ TClienteIterator }
constructor TClienteIterator.Create(const List: TClienteList);
begin
 inherited Create();
 self._list := List;
 self.Reset();
end;
function TClienteIterator.CurrentItem(): TCliente;
begin
 if (self._list <> nil) and ((self._currentItem >= 0) and
   (self._currentItem < self._list.GetCount)) then
   result := _list.GetItem(self._currentItem)
 else
   result := nil;
end;
function TClienteIterator.Next(): Boolean;
begin
 result := (self._list <> nil) and
   (self._currentItem < (self._list.GetCount - 1));
 if result then
   Inc(self._currentItem);
end;
procedure TClienteIterator.Reset();
begin
 self._currentItem := -1;
end;

{ TCliente }
constructor TCliente.Create(ID: integer; Nome: string);
begin
 self._id := ID;
 self._nome := Nome;
end;
function TCliente.GetID(): integer;
begin
 result := self._id;
end;
function TCliente.GetNome(): string;
begin
 result := self._nome;
end;
procedure TCliente.SetID(const Value: integer);
begin
 self._id := Value;
end;
procedure TCliente.SetNome(const Value: string);
begin
 self._nome := Value;
end;
{ TClienteList }
procedure TClienteList.Add(const Item: TCliente);
begin
 self._items.Add(item);
end;
procedure TClienteList.Clear();
begin
  self._items.Clear();
end;
function TClienteList.Contains(const Item: TCliente): Boolean;
begin
 result := self._items.IndexOf(item) >= 0;
end;
constructor TClienteList.Create();
begin
 self._items := TObjectList.Create();
end;
procedure TClienteList.Delete(Idx: Integer);
begin
 self._items.Delete(Idx);
end;
destructor TClienteList.Destroy();
begin
 self._items.Free();
 inherited;
end;
function TClienteList.GetCount(): Integer;
begin
 result := self._items.Count;
end;
function TClienteList.GetItem(Idx: Integer): TCliente;
begin
 result := self._items[Idx] as TCliente;
end;
procedure TClienteList.Insert(Idx: Integer; const Item: TCliente);
begin
 self._items.Insert(Idx,Item);
end;
procedure TClienteList.Remove(const Item: TCliente);
begin
 self._items.Remove(item);
end;
end.
```

Listagem 6. Usando um Iterator

```
program IteratorApp;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

function CriaClientes(): TClienteList;
begin
  result := TClienteList.Create();
  result.Add(TCliente.Create(1,'Guinther'));
  result.Add(TCliente.Create(2,'Rudolfo'));
  result.Add(TCliente.Create(3,'Wesley'));
  result.Add(TCliente.Create(4,'Erlei'));
  result.Add(TCliente.Create(5,'Paulo'));
  result.Add(TCliente.Create(6,'Daniel'));
end;

var
  list: TClienteList;
  itr: TClienteIterator;
begin
  list := CriaClientes();
  try
    itr := TClienteIterator.Create(list);
    try
      while itr.Next() do
        WriteLn(itr.CurrentItem.Nome);
    finally
      itr.Free();
    end;
  finally
    list.Free;
  end;
  ReadIn;
end.
```

Listagem 7. Mediator

```
unit uFramework;
interface
type
  Mediator = class; //forward

  // Colleague (abstract)
  Colega = class abstract
    protected _mediator: Mediator;
    //Construtor
    public constructor Create (mediator: Mediator);
  end;

  // Concrete Colleague
  Suporte = class(Colega)
    public constructor Create(mediator: Mediator);
    public procedure Enviar(mensagem: string);
    public procedure Notificar(mensagem: string);
  end;

  // Concrete Colleague
  Usuario = class(Colega)
  public
    public constructor Create(mediator: Mediator);
    public procedure Enviar(mensagem: string);
    public procedure Notificar(mensagem: string);
  end;

  // Mediator
  Mediator = class abstract
    public procedure Enviar(mensagem: string; colega: Colega); virtual; abstract;
  end;

  // Concrete Mediator
  ConcreteMediator = class(Mediator)
    private _suporte: Suporte;
    private _usuario: Usuario;
    public property Suporte: Suporte write _suporte;
    public property Usuario: Usuario write _usuario;
    public procedure Enviar(mensagem: string; colega: Colega); override;
  end;

implementation

// Concrete Mediator
procedure ConcreteMediator.Enviar(mensagem: string; colega: Colega);
begin
  if colega = _usuario then
    _suporte.Notificar(mensagem)
  else
    _usuario.Notificar(mensagem);
end;

// Abstract Colleague (Colega)
constructor Colega.Create(mediator: Mediator);
begin
  self._mediator := mediator;
end;

// Concrete Colleague (Suporte)
constructor Suporte.Create(mediator: Mediator);
begin
  inherited Create(mediator);
end;

procedure Suporte.Enviar(mensagem: string);
begin
  _mediator.Enviar(mensagem, self);
end;

procedure Suporte.Notificar(mensagem: string);
begin
  WriteLn('Suporte recebeu a mensagem: ' + mensagem);
end;

// Concrete Colleague (Usuario)
constructor Usuario.Create(mediator: Mediator);
begin
  inherited Create(mediator);
end;

procedure Usuario.Enviar(mensagem: string);
begin
  _mediator.Enviar(mensagem, self);
end;

procedure Usuario.Notificar(mensagem: string);
begin
  WriteLn('Usuário recebeu a mensagem: ' + mensagem)
end;

end.
```

uma mensagem recebida pelo mesmo a partir de outro colega. Quando precisam comunicar uma com a outra, não fazem isso diretamente (o que causaria um acoplamento), fazem isto através de um mediador (Mediator). Esse procedimento pode ser visto no método Enviar, que é implementado por ambas as classes. O que o método faz é simplesmente delegar (veja, mais uma vez delegação) a tarefa de repassar uma mensagem para o mediador, que fará um teste de quem quer comunicar quem, entregando a mensagem à parte interessada. A **Listagem 8** mostra um exemplo do uso do framework, onde um mediador é usado para trocar uma mensagem (chat) entre usuário e suporte, sem que para isso eles se enxerguem. Observe na **Figura 4** o padrão Mediator no diagrama de UML.

De certa forma esse padrão é muito parecido com um Command, pois ele também tem o objetivo de desacoplar classes concretas que seguem suas próprias especificações. A diferença aqui fica por conta da presença do mediador, que conhece ambas as partes. No Command ninguém conhece ninguém, apenas a abstração do comando. Em Mediator, não há abstração de um comando ou método a ser invocado por qualquer uma das partes (que é uma comunicação bidirecional). O mediador pode chamar qualquer método de cada uma das partes envolvidas, que agem tanto como receiver e invoker (semelhante ao Command). Também podemos comparar o Mediator a uma espécie de Adapter, já que ele adapta interfaces distintas para se comunicarem. A diferença novamente fica no fato de que as classes "colegas" não precisam descender de uma classe base abstrata, pois, diferente de um Adapter, uma destas classes não precisa substituir a outra em uma comunicação, de forma a tornar sua interface compatível com a outra. Aqui usuário é uma coisa, suporte é outra.

Temos outro exemplo típico na vida real onde um mediador é essencial. Imagine que uma torre de controle de um aeroporto precisa controlar todo o seu tráfego aéreo. As aeronaves (colegas) precisam utilizar essa torre como mediador, a fim de evitar que uma aeronave tente pousar ao mesmo tempo que outra está levantando voo. As aeronaves muitas vezes não se comunicam entre si, seja por questões de segurança, controle, ou até mesmo por serem de companhias e países diferentes (com idiomas diferentes). A presença do mediador é crucial para coordenar a comunicação entre todos estes objetos complexos, a fim de garantir o funcionamento do sistema.

Memento

A intenção do padrão Memento é, sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto, para que este possa ser restaurado para este estado mais tarde. O padrão pode ser aplicado quando: o estado de um objeto deve ser salvo de maneira que possa ser restaurado para esse mesmo estado mais tarde; quando uma interface do estado exporia detalhes de implementação e romperia o encapsulamento do objeto. Benefícios: preservação das fronteiras de encapsulamento; definição de interfaces mínimas e amplas.

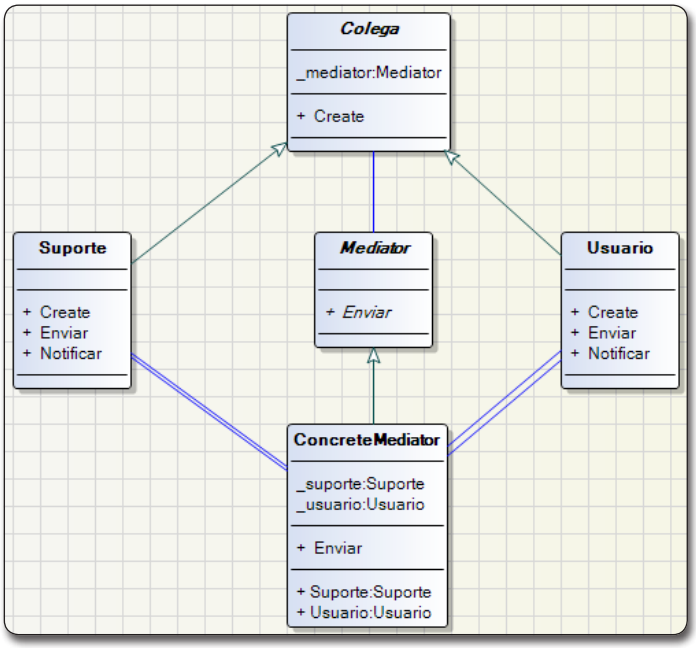

Figura 4. Estrutura de classes do exemplo usando Mediator

Listagem 8. Usando um Mediator

```
program MediatorApp;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  // Mediador
  mediator: ConcreteMediator;
  // Colegas
  sup: Suporte;
  user: Usuario;
begin
  // Cria mediador
  mediator := ConcreteMediator.Create();
  // Cria colega (suporte) e configura quem será o mediador
  sup := Suporte.Create(mediator);
  // Cria colega (usuário) e configura quem será o mediador
  user := Usuario.Create(mediator);
  // Informa ao mediador quem serão as partes envolvidas
  mediator.Suporte := sup;
  mediator.Usuario := user;
  // Faz a comunicação entre as partes usando mediador
  user.Enviar('Meu Windows não está iniciando');
  sup.Enviar('Formate a máquina');
  ReadLn;
end.
```

Participantes:

• Memento – armazena o estado interno do objeto Originador. O Memento pode armazenar um pouco ou muito do estado do Originador, conforme necessário; Protege contra acessos por objetos que não o Originador;

• Originator – cria um Memento contendo um instantâneo do seu estado interno corrente; usa o Memento para restaurar o seu estado interno;
• Caretaker – é responsável pela custódia do Memento; nunca opera ou examina os conteúdos de um Memento;

A **Listagem 9** mostra um framework que implementa o padrão Memento. Aqui temos uma classe chamada Pessoa, que possui um determinado estado, representado por uma propriedade. Poderíamos utilizar outras propriedades, como nome, endereço, telefone etc. Esta classe tem um método chamado CreateMemento, que nada mais faz do que criar um objeto do tipo Memento, que simplesmente guardará seu estado. Aqui a palavra Memento faz jus a sua tradução (memória, em português), pois simplesmente faz o papel de uma "variável auxiliar", que permite guardar e recuperar o estado de um outro objeto a qualquer momento. Esta memória é guardada em um outro participante do padrão, o Caretaker (ou guarda, zelador, em português), a fim de criar uma abstração entre Memento e objeto a ser persistido em memória (ou outro meio de armazenamento). A **Listagem 10** mostra a criação do objeto Pessoa, a gravação e recuperação do seu estado. A **Figura 5** exibe a representação do padrão Memento.

Note que, para realizar este trabalho de recuperação do estado de um objeto, a partir de outro, seria necessário que o objeto que precisa armazenar o outro, conhecesse totalmente sua interface, propriedades e atributos públicos. Deveria ser capaz de ler estaticamente cada um, causando um acoplamento enorme entre ambas as classes, quase indivisível e sem possibilidade reutilização. Além disso, se todo o estado do objeto precisasse ser guardado, incluindo atributos privados internos, teríamos uma violação séria de encapsulamento. Utilizar reflexão neste caso também seria algo inapropriado.

Podemos observar a aplicação do Memento em alguns frameworks de persistência, ou mesmo Web. Em sistemas Web state-less, como o ASP.NET, o estado de um formulário precisa ser mantido entre requisições, para que possa ser restaurado quando algo precisa ser processado no servidor. Nesse caso, o .NET Framework utiliza o padrão Memento para "memorizar" o estado de um formulário em um Caretaker, que lá no ASP.NET recebe o nome de ViewState. O formulário é então destruído após a requisição, mas pode ser reconstruído a partir do ViewState. De forma semelhante e em termos práticos, podemos tratar a própria propriedade Data do ClientDataSet como um Memento. De fato, os dados são obtidos a partir de um DataSetProvider, que obtém isso através de IProviderSupport de algum DataSet de algum engine de acesso a dados como DBExpress ou FireDac. Data é a memória dos dados, de forma que as conexões podem ser todas fechadas e o trabalho realizado off-line em memória.

Listagem 9. Memento

```
unit uFramework;

interface

type
 // Memento
 Memento = class
   private _state: string;
   public constructor Create(state: string);
   property State: string read _state;
 end;

 // Originator
 Pessoa = class
   private _state: string;
   public property State: string read _state write _state;
   public function CreateMemento(): Memento;
   public procedure setMemento(memento: Memento);
 end;

 // Caretaker
 Caretaker = class
   private _memento: Memento;
   public property Memento: Memento read _memento write _memento;
 end;

implementation

// Pessoa
function Pessoa.CreateMemento(): Memento;
begin
  result := Memento.Create(_state);
end;

procedure Pessoa.setMemento(memento: Memento);
begin
  WriteLn('Restaurando o estado...');
  State := memento.State;
end;

// Memento
constructor Memento.Create(state: string);
begin
  self._state := state;
end;

end.
```

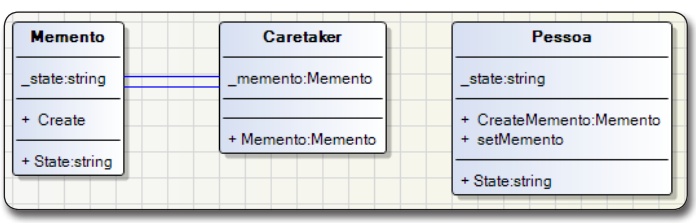

Figura 5. Estrutura de classes do exemplo usando Memento

Proxy

A intenção do padrão Proxy é fornecer um substituto ou espaço reservado para outro objeto para controlar o acesso a ele. Um proxy, em sua forma mais geral, é uma classe que funciona como uma interface para outra classe. A classe proxy poderia conectar-se a qualquer coisa: uma conexão de rede, um objeto grande em memória, um arquivo, ou algum recurso que é difícil ou impossível de ser duplicado. Entre os benefícios podemos citar: um proxy pode ocultar o fato de que um objeto reside em um espaço de memória diferente; um proxy virtual pode executar otimizações, tais como criação de objetos sob demanda.

Listagem 10. Usando um Memento

```
program MementoApp;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  gp: Pessoa;
  c: Caretaker;
begin
  // cria originator
  gp := Pessoa.Create();
  gp.State := 'Guinther';
  // mostra estado original (criado)
  WriteLn('Estado original: ' + gp.State);
  // cria Caretaker
  c := Caretaker.Create();
  c.Memento := gp.CreateMemento();
  // trocando o estado...
  gp.State := 'Pauli';
  // mostra estado atual
  WriteLn('Estado atual: ' + gp.State);
  // restaurar o estado original
  gp.setMemento(c.Memento);
  // mostra estado atual
  WriteLn('Estado restaurado: ' + gp.State);
  ReadLn;
end.
```

Participantes:

• Proxy – mantém uma referência que permite ao proxy acessar o objeto real; fornece uma interface idêntica ao de Subject, por esse motivo ele pode substituir o objeto real (Real Subject); controla o acesso ao objeto real, podendo ser responsável pela criação e destruição;

• Subject – define uma interface comum para RealSubject e Proxy;

• RealSubject – define o objeto real que o proxy representa;

Quando importamos um Web Service SOAP em Delphi, estamos criando uma classe proxy. Neste caso, quando importamos uma WSDL, com base nas informações XML que descrevem o serviço SOAP (Web Service), é criada uma classe no cliente que possui os mesmos métodos da classe no servidor. Você pode então chamar qualquer método dessa classe cliente (o proxy), que internamente por delegação repassará as chamadas ao objeto remoto. O Visual Studio quando trabalha com serviços do Windows Communication Foundation (WCF), também gera classes proxy para permitir essa comunicação entre objetos remotos. Mas não é só em ambientes distribuídos que podemos ver a implementação deste importante padrão de projeto, também conhecido como Wrapper. Por exemplo, quando importamos um Type Library para trabalhar com um determinado objeto COM, MTS, COM+, ActiveX ou via OLE Automation (Word, Excel, PDF, Flash), fazendo assim a interoperabilidade entre aplicações Delphi e outros aplicativos escritos em outras linguagens, temos a opção de gerar um objeto Proxy. Além de servir para gerar toda a interface do objeto que queremos consumir, ele encapsula o acesso a esses objetos de forma interna, passando essas chamadas por delegação (note mais uma vez, outro padrão que usa delegação). Normalmente o Proxy também controla a instância do objeto, semelhante ao que faz o Singleton. A paleta Servers do Delphi (surgida lá no Delphi 5), nada mais é do que a implementação de dezenas de classes proxy. Antes disso, quando se precisava trabalhar com DLLs, era necessário redeclarar manualmente cada método em uma sessão do formulário ou unit, usando cláusulas external. O próprio DataSnap novo também faz uso de Proxy, porém, dando um outro nome. Quando você importa a definição de um Server Module, para consumir seus métodos remotamente (que não sejam de IAppServer), é gerado um Client Module (que é como um Proxy). Isso também permite uma ligação estática no seu código, para a correta chamada de métodos, o que chamamos de early-binding. Antes disso, utilizava-se algumas vezes como opção o late-binding, chamando dinamicamente métodos de AppServer de um Remote Server, sem apoio de um proxy ou mesmo do Code Insight. Dessa forma, um padrão Proxy também permite a verificação de tipos em tempo de compilação, antes que essas informações cheguem de forma errada ao seu destino, por exemplo, um servidor remoto.

As **Listagens 11** e **12** mostram um exemplo do uso do padrão Proxy. Fiz um exemplo bastante simples, já que o padrão por si próprio é bastante simples de entender e possui poucos participantes. Temos primeiramente a definição de uma interface de uma calculadora que processa operações matemáticas, a qual deve ser implementada tanto pelo objeto remoto (RealSubject), quando pela classe Proxy. Ou seja, eles têm interfaces idênticas, qualquer método que existir no alvo pode ser chamado localmente (desde que exportado e público). O Proxy usa delegação para repassar cada uma das chamadas a um objeto interno privado. Note que com isso, ganhamos a força do encapsulamento para alterar a qualquer momento quem é nosso objeto destino, podemos trocar de um Web Service Win32 para um Web Service .NET sem afetar a classe cliente. De fato, este é o objetivo da maioria dos padrões de projeto, deixar sistemas preparados para mudanças, apoiando-se muito na abstração, encapsulamento e delegação. Por último, a **Figura 6** ilustra os participantes do padrão Proxy.

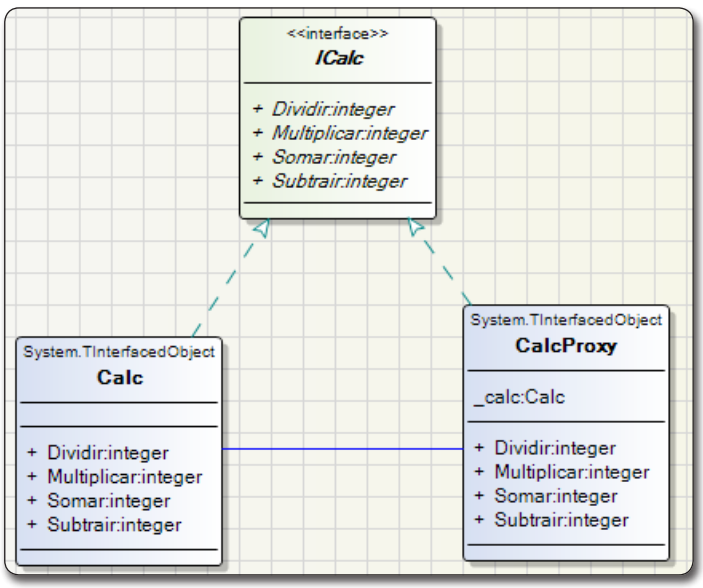

Figura 6. Estrutura de classes do exemplo usando Proxy

Listagem 11. Proxy

```
unit uFramework;
interface
type
  // Subject
  ICalc = interface
    function Somar(x,y: integer): integer;
    function Subtrair(x,y: integer): integer;
    function Dividir(x,y: integer): integer;
    function Multiplicar(x,y: integer): integer;
  end;
  // RealSubject
  Calc = class(TInterfacedObject,ICalc)
    public function Somar(x,y: integer): integer;
    public function Subtrair(x,y: integer): integer;
    public function Dividir(x,y: integer): integer;
    public function Multiplicar(x,y: integer): integer;
  end;
  // Proxy
  CalcProxy = class(TInterfacedObject,ICalc)
    private _calc: Calc;
    public function Somar(x,y: integer): integer;
    public function Subtrair(x,y: integer): integer;
    public function Dividir(x,y: integer): integer;
    public function Multiplicar(x,y: integer): integer;
  end;
implementation
{ Calc }
function Calc.Dividir(x, y: integer): integer;
begin
  // considerando apenas inteiros
  result := trunc(x / y);
end;
function Calc.Multiplicar(x, y: integer): integer;
begin
  result := x * y;
end;
function Calc.Somar(x, y: integer): integer;
begin
  result := x + y;
end;
function Calc.Subtrair(x, y: integer): integer;
begin
  result := x - y;
end;
{ CalcProxy }
function CalcProxy.Dividir(x, y: integer): integer;
begin
  // instancia por demanda (lazy)
  if (_calc = nil) then
    _calc := Calc.Create();
  // repassando chamada ao RealSubject
  // Delegação
  result := _calc.Dividir(x, y);
end;
function CalcProxy.Multiplicar(x, y: integer): integer;
begin
  // instancia por demanda (lazy)
  if (_calc = nil) then
    _calc := Calc.Create();
  // repassando chamada ao RealSubject
  // Delegação
  result := _calc.Multiplicar(x, y);
end;
function CalcProxy.Somar(x, y: integer): integer;
begin
  // instancia por demanda (lazy)
  if (_calc = nil) then
    _calc := Calc.Create();
  // repassando chamada ao RealSubject
  // Delegação
  result := _calc.Somar(x, y);
end;
function CalcProxy.Subtrair(x, y: integer): integer;
begin
  // instancia por demanda (lazy)
  if (_calc = nil) then
    _calc := Calc.Create();
  // repassando chamada ao RealSubject
  // Delegação
  result := _calc.Subtrair(x, y);
end;
end.
```

Listagem 12. Usando um Proxy

```
program Proxy;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  c: ICalc; // programe para uma interface/abstração
  r: integer;
begin
  // Usando Proxy
  c := CalcProxy.Create();
  r := c.Somar(3,5);
  Writeln(IntToStr(r));
  r := c.Multiplicar(5,4);
  Writeln(IntToStr(r));
  r := c.Subtrair(10,3);
  Writeln(IntToStr(r));
  r := c.Dividir(9,3);
  Writeln(IntToStr(r));
  Readln;
end.
```

Conclusão

Neste artigo vimos alguns padrões de projeto da classe comportamentais: Chain of Responsibility, Command, Iterator, Mediator e Memento. Estes padrões ajudam a tornar sistemas de software mais fáceis de serem mantidos e evoluídos. Como vimos, eles são destinados a resolver problemas com operações, comunicação e o relacionamento entre objetos complexos, que comprometem a reutilização de módulos do sistema e aumentam o acoplamento. Chain of Responsibility permite criar uma cadeia de responsabilidades e passar operações entre objetos encadeados, sem acoplamento. Command permite trocar um comando a ser executado entre invocador e receptor, também sem acoplamento entre eles. Iterator é um padrão famoso que permite a iteração entre itens de uma classe, por exemplo, uma coleção, como mostramos no exemplo. Mediator também facilita a comunicação entre objetos sem causar acoplamento, nesse caso, objetos complexos. Já o Memento não fere o encapsulamento para obter o estado de um objeto. Também estudamos mais um padrão da categoria estrutural, o Proxy.

Autor

Guinther Pauli
http://www.facebook.com/guintherpauli
http://www.twitter.com/guintherpauli
http://br.linkedin.com/in/guintherpauli
http://www.gpauli.com
http://guintherpauli.blogspot.com.br
http://www.devmedia.com.br/guintherpauli

Arquiteto de Software certificado pela Microsoft, Consultor, Instrutor, Autor, Palestrante, Professor de Graduação e de Pós-Graduação, Colaborador Editorial das Revistas .NET Magazine Brasil e ClubeDelphi, Certificado oficial Microsoft MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET, C# e Arquitetura) - Certificado oficial Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Kylix. Most Valuable Professional (MVP) - Embarcadero Technologies. Entre os principais projetos que participou, foi arquiteto e desenvolvedor do sistema de doações on-line do projeto Criança Esperança da Rede Globo / Unesco. Beta Test de várias versões do Delphi, palestrante em 5 edições da Borland Conference, maior evento de desenvolvedores do país. Ministra treinamentos corporativos in-company, já tendo treinado profissionais de várias empresas dos estados do RS, SC, PR, SP, RJ, GO, TO, DF, MG, AM e PE. Autor de mais de 1100 videoaulas de ensino a distância pela DevMedia, publicou mais de 130 artigos, além do livro Delphi Programação para Banco de Dados e Web . Fez a cobertura de grandes eventos no exterior, como o TechEd da Microsoft e a BorCon da Borland americana. Experiência em Delphi de 18 anos e 13 anos em .NET / C#.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Criação de relatórios via WebServices e DataSnap

Expondo para as mais diversas plataformas

A criação de relatórios em aplicações corporativas acaba na maioria dos casos sendo uma preocupação a mais ao desenvolvedor principalmente se ele estiver usando a tecnologia DataSnap em aplicações distribuídas. Por mais que haja ferramentas que possibilitem a criação destes relatórios para cada uma das aplicações clientes que acessam o servidor de aplicação DataSnap, em muitos casos as habilidades de gerenciar a ferramenta de geração de relatórios, o custo de uma licença de uso e até mesmo a redundância de implementação do mesmo tipo de relatório, podem se tornar um empecilho de utilização da mesma.

Para esta situação encontramos a solução através de um Web Service que é utilizado para integração e comunicação entre diferentes aplicações. Assim sendo, por meio deste Web Service será gerado um relatório em Rave Reports e disponibilizado como produto final apenas um arquivo em pdf para ser mostrado por qualquer aplicação cliente.

Para desenvolver toda esta estrutura, definiremos primeiramente a arquitetura da aplicação, passando em seguida para a implementação da mesma através da criação da base de dados, servidor de aplicação DataSnap, Web Service e finalmente três exemplos de aplicações clientes que possam exemplificar com tranquilidade a solução proposta neste artigo.

Resumo DevMan

Porque esse artigo é útil:

Neste artigo demonstraremos de forma prática como é possível utilizar Web Services para disponibilizar relatórios RAVE, em soluções multicamadas, desenvolvidas sobre o Framework DataSnap do Delphi XE, a múltiplos clientes, de tecnologias heterogêneas Web e Desktop tais como VCL Forms, PHP e ASP .NET. É apresentada aos desenvolvedores de aplicações baseadas na arquitetura DataSnap a possibilidade de criar relatórios apenas no lado servidor e disponibilizá-los para as aplicações clientes através de Web Services. A arquitetura DataSnap permite a comunicação entre a aplicação servidora e as várias aplicações clientes através do protocolo TCP/IP e HTTP. A comunicação entre cliente e servidor de aplicação se dá através da chamada de métodos no servidor. Assim sendo, a criação de relatórios para esta arquitetura acaba na maioria das vezes sendo uma preocupação a mais para o desenvolvedor, pois dependendo da plataforma de desenvolvimento da aplicação cliente, existem várias opções de disponibilizar e desenvolver relatórios. Com o modelo apresentado neste artigo o desenvolvedor vai se preocupar em criar o seu relatório apenas no servidor, disponibilizá-lo através de um Web Service e a aplicação cliente terá a única preocupação de consumir este Web Service, aumentando assim a produtividade do desenvolvimento e até mesmo a migração de software cliente-servidor para a arquitetura DataSnap.

Arquitetura da aplicação

A criação de relatórios em aplicações cliente que usam a arquitetura DataSnap é um desafio aos desenvolvedores, pois estes devem escolher frameworks, componentes e outros métodos de geração de relatórios para as mesmas, partindo do premissa que tendo um servidor de aplicação DataSnap é possível implementar aplicações clientes na maioria das plataformas e linguagens de programação existentes que suportam RPC (Remote Procedure Call) ou comunicação através de REST (Representational State Transfer).

Assim sendo, pensando neste desafio e na possibilidade otimizar tempo de desenvolvimento e organizar melhor os relatórios a serem disponibilizados em servidores de aplicação DataSnap criou-se uma arquitetura com este objetivo, conforme mostra a **Figura 1**.

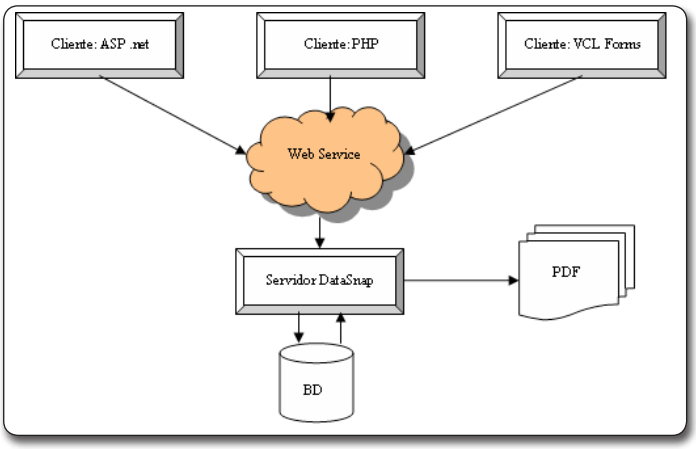

Figura 1. Arquitetura da aplicação

Nesta arquitetura cada aplicação cliente terá a funcionalidade de invocar o Web Service responsável pela geração de relatórios. Este Web Service ao ser invocado fará uma chamada remota ao Servidor DataSnap que terá a responsabilidade de gerar o relatório, salvá-lo no formato pdf ou qualquer outro formato que seja transparente as aplicações clientes como XML, JSON e tantos outros. Ficando por conta de cada uma das aplicações cliente apenas criar o mecanismo de mostrar este relatório dentro da aplicação de acordo com o formato de relatório escolhido.

Organizando o projeto

Após conhecermos a arquitetura da aplicação que é justamente utilizar Web Services para disponibilizar relatórios, em soluções multicamadas, desenvolvidas sobre o Framework DataSnap do Delphi XE, a múltiplos clientes, de tecnologias heterogêneas Web e Desktop, tais como VCL Forms, PHP e .NET, vamos organizar a estrutura de pastas do projeto a ser desenvolvido, conforme mostra a **Figura 2**. Na pasta "Banco" iremos guardar o banco de dados da aplicação de exemplo, na pasta "Cliente" todos as aplicações clientes separadas por tipo de plataforma de desenvolvimento, na pasta "Servidor" todos os arquivos referente ao servidor de aplicação DataSnap, já na pasta "WebService" os arquivos de projeto do Web Service e por fim todos o output dos arquivos executáveis deverão ser configurados para caírem dentro da pasta "bin".

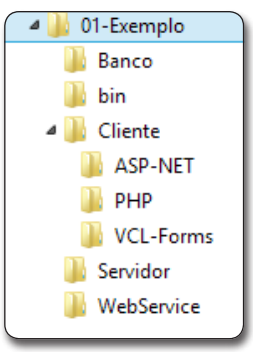

Figura 2. Organização do projeto

Criando o Banco de Dados

Para começar é necessário criar banco de dados usando o Firebird 1.5 salvando-o na pasta "Banco" (**Figura 2).** Para o exemplo prático será criada apenas uma tabela no banco de dados (**Listagem 1**). Seria interessante popular esta tabela com no mínimo 10 registros.

Listagem 1. Criação da tabela de exemplo

```
01 CREATE TABLE TB_CAD_ALUNO (
02   MATRICULA VARCHAR(10) NOT NULL,
03   NOME_ALUNO VARCHAR(100) NOT NULL,
04   ENDERECO VARCHAR(200),
05   TELEFONE VARCHAR(20),
06   CPF VARCHAR(11)
07 );
08 ALTER TABLE TB_CAD_ALUNO ADD CONSTRAINT TB_CAD_ALUNO_PK
     PRIMARY KEY (MATRICULA);
```

Servidor de Aplicação DataSnap

Atualmente no Delphi XE, a tecnologia responsável por prover a construção de aplicações multicamadas é o DataSnap, que possui como base o dbExpress. Assim sendo, o uso do DataSnap no desenvolvimento de aplicações multicamadas está muito relacionado à utilização de outros dois elementos, o dbExpress e o TClientDataSet. O primeiro, para a parte de acesso tanto de dados quanto ao Servidor de Aplicação e o segundo para o acesso aos dados no lado Cliente.

Um servidor de aplicação DataSnap nada mais é do que uma aplicação Delphi tradicional, contendo alguns elementos-chave adequados ao modelo multicamadas. Sendo assim, ela pode ser de três tipos: VCL Forms Application, Console Application ou Service Application. A escolha por uma delas varia de acordo com a necessidade e objetivo do projeto.

Para implementar a camada da arquitetura da aplicação referente ao servidor de aplicação DataSnap, vamos criar no Delphi XE uma nova aplicação DataSnap Server do tipo VCL Forms Applications usando o TDSServerModule como o tipo ancestral para as classes dos métodos servidores. Salve a aplicação DataSnap Server dentro da pasta "Servidor" (**Figura 2),** e mude apenas o nome do projeto para Servidor.

Depois de criada a aplicação observe na Unit ServerContainterUnit1 os elementos chave, no lado servidor, sendo os principais itens para o funcionamento do DataSnap como:

- DSServer: principal componente de configuração do servidor DataSnap, responsável, dentre outras coisas, pela gerência de classes do servidor;
- DSServerClass: componente usado para expor os recursos públicos de uma classe do lado servidor para clientes remotos, sendo também responsável por determinar o ciclo de vida de cada instância desta classe;
- TDSTCPServerTransport: componente que habilita o transporte baseado em soquete (TCP) ao servidor.

Finalmente, configuramos o projeto também para que a pasta de destino do binário (.exe) seja a pasta "bin" (**Figura 2).** Agora é rodar a aplicação servidora e pronto. Já se tem neste o servidor de aplicação DataSnap em funcionamento (**Figura 3**).

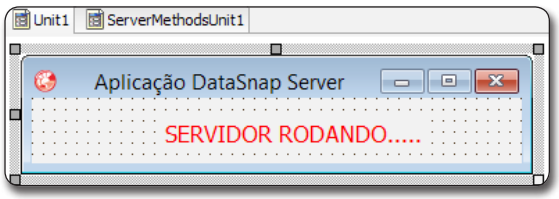

Figura 3. Tela inicial da aplicação DataSnap Server

Acesso a dados

Toda a parte de acesso a dados se concentra também no servidor de aplicação. Em termos de componentes, pouca coisa muda em relação as tradicionais aplicações Client/Server, pois somente o componente TClientDataSet, que não estará mais presente no servidor de aplicação, sendo este transferido para as aplicações clientes. Assim sendo, é necessário se estabelecer uma forma de expor os DataSets, via TDataSetProvider, ao respectivo ClientDataSet (aplicação cliente).

Uma solução para isso é usar um novo container, particular ao DataSnap, o TDSServerModule, que também descente de TDataModule mas que contempla todas as características adicionais necessárias para se expor DataSets para as aplicações clientes.

Desta forma é necessário customizar a aplicação servidora para que esta possa se conectar com a Base de Dados. É necessário colocar um componente do tipo TSQLConnection no ServerMethodsUnit1, conforme mostra a **Figura 4,** e além do mais, configurá-lo para acesso ao banco de dados criado neste artigo.

Para que possamos ilustrar através de nosso exemplo o acesso ao banco de dados é necessário colocar e configurar um componente TSQLDataSet no ServerMethodsUnit1 conforme mostra a **Figura 5**. Para configurar o TSQLDataSet é só setar a propriedade SQLConnection deste para a DBXCONEXAO e criar na propriedade CommandText a consulta às informações da tabela criada conforme **Listagem 1**. Depois de configurado o TSQLDataSet é necessário implementar o método getAllAlunos que irá disponibilizar através de um TDataSet as informações às aplicações clientes (**Listagem 2**).

Neste ponto, o componente TDSServerClass expõe uma classe do servidor aos clientes remotos através do evento OnGetClass, apontando para um parâmetro denominado Persistent-Classs e consequentemente o método getAllAlunos implementado já está disponível as aplicações clientes que queriam fazer uso deste.

Listagem 2. Método getAllAlunos:TDataSet

```
01 function TServerMethods1.getAllAlunos:TDataSet;
02 begin
03   getAlunos.Open;
04   result := getAlunos;
05 end;
```

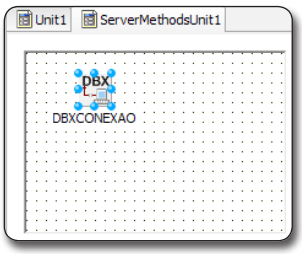

Figura 4. Componente de conexão com o banco de dados

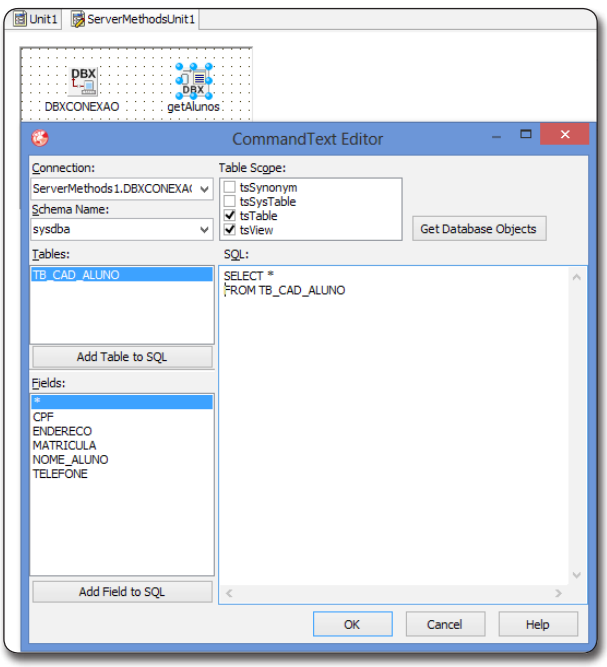

Figura 5. Configuração do TSQLDataSet

Web Service

Web service é uma das soluções para integração e comunicação entre diferentes aplicações. Por meio de um Web service é possível disponibilizar funções ou mensagens para qualquer outra aplicação na internet ou em aplicações corporativas internas. Neste contexto, o Delphi XE nos fornece o protocolo SOAP (Simple Object Access Protocol) para a criação de Web services.

SOAP é um protocolo de comunicação baseado em XML quer permite que aplicações troquem informações utilizando o protocolo HTTP. Mais especificamente, SOAP é um protocolo para acessar um Web service e por seguir o padrão W3C, é independente de plataforma e de linguagem de programação.

Para conhecer os serviços ou métodos disponibilizados pelo Web Service, é utilizado um documento XML padronizado de acordo com o WSDL (Web Service Description Language). Ele disponibiliza detalhes do funcionamento dos serviços e normalmente é público. Para ser acessado e identificado, assim como outros recursos da Internet, o Web Service possui uma URL (Uniform Resource Locator), que é simplesmente o endereço do mesmo. Veja em "Entendendo os Padrões de Descrição de Web Services" (ver seção **Links**).

Assim sendo, para este exemplo iremos criar no Delphi XE uma nova aplicação Web Services do tipo SOAP Server Application escolhendo o tipo de serviço web como ISAPI, conforme mostra a **Figura 6**. Salve a aplicação Web Service dentro da pasta "WebService" **(Figura 2),** e mude apenas o nome do projeto para WebService.

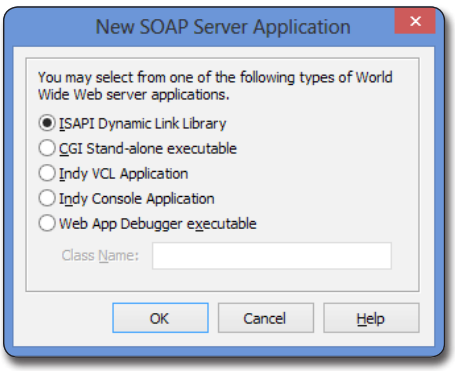

Figura 6. Criação do Web Service

Depois é preciso criar um SOAP Data Module com o nome dmConexaoRave, conforme mostra a **Figura 7**, salve a Unit criada. Neste exato momento já se tem o Web Service criado com a interface IdmConexaoRave responsável por disponibilizar os métodos a serem consumidos através do Web Service.

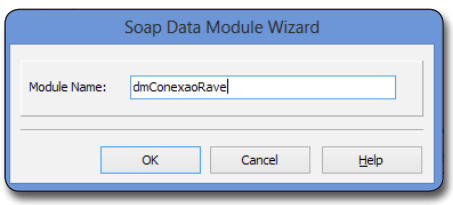

Figura 7. SOAP Data Module

Como o objetivo do artigo é disponibilizar relatórios através deste Web Service será necessário implementar o acesso ao servidor de aplicação DataSnap configurando o TSQLConnection e gerando as DataSnap client classes, ou seja, a classe proxy de comunicação com o servidor de aplicação DataSnap, para que seja possível usar o método getAllAlunos(), responsável por disponibilizar um DataSet com as informações presentes no banco de dados.

Além do que de posse do DataSet será necessário colocar e configurar alguns componentes do Rave Reports (TRvProject, TRVSystem, TRVRenderPDF, TRVRenderHTML, TRVDataSetConnection, TClientDataSet, TDataSetProvider) no SOAP Data Module conforme mostra a **Figura 8**. Para que seja possível criar o relatório em Rave será necessário criar manualmente os TFields do componente TClienteDataSet (cdsRelAlunos) para que o Rave possa montar o detalhe do relatório. Assim sendo, o próximo passo é criar no Rave o relatório com o nome de prsRelAlunos.rav, conforme mostra a **Figura 9**.

Neste ponto precisamos criar no SOAP Data Module o método que irá buscar os dados dos alunos, gerar o relatório do Rave e exportar para um arquivo PDF, que se encontra na **Listagem 3** comentado de acordo com a ação de cada bloco de código.

Assim sendo, as aplicações clientes após consumirem o Web Service deverão disponibilizar uma forma de mostrar o PDF do relatório gerado pelo Web Service. Finalmente, para que o Web Service seja consumido por alguma aplicação é necessário publicá-lo em algum servidor web.

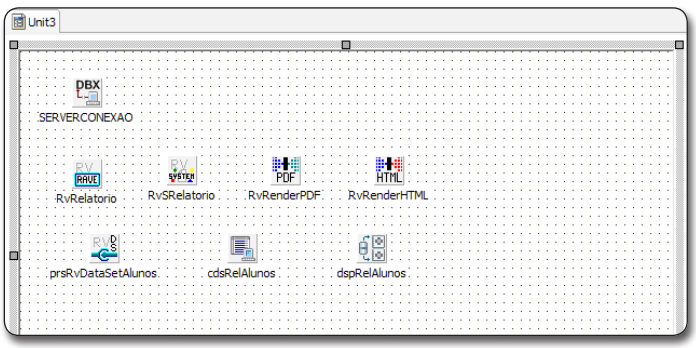

Figura 8. Componentes de conexão com o servidor DataSnap e geração de relatórios em Rave Reports

Figura 9. Relatório no Rave - prsRelAlunos.rav

Listagem 3. Método TdmConexaoRave.ExecutaRelatorio:Boolean;

```
01 function TdmConexaoRave.ExecutaRelatorio:Boolean; stdcall;
02 begin
03  inherited;
04  result := false;
05  try
06   //cria a conexão com a classe cliente
07   serverClasses := TServerMethods1Client.Create
     (SERVERCONEXAO.DBXConnection);
08   //busca os alunos
09   cdsRelAlunos.Close;
10   dspRelAlunos.DataSet := serverClasses.getAllAlunos;
11   cdsRelAlunos.Open;
12   //chamar o relatório projeto do rave
13   RvRelatorio.ProjectFile := gsAppPath + 'prsRelAlunos.rav';
14   //Define o path do PDF para exportação
15   RvSRelatorio.DoNativeOutput:= false;
16   RvSRelatorio.RenderObject := RvRenderPDF;
17   RvSRelatorio.OutputFileName:= gsAppPath + 'getAllAlunos.pdf';
18   //Executa o relatório e faz com ele seja exportado
19   RvRelatorio.Execute;
20   result := true;
21   finally
22   serverClasses.Free;
23  end;
24 end;
```

A publicação de Web Services com finalidade de criar relatórios não requer necessariamente que seja hospedada em um servidor com acesso a internet, sendo que a mesma poderá ser instalada de forma local, ou seja, apenas estará disponível para a aplicação que deverá consumi-la, mas nada impede que esta seja distribuída pela Web a fim de ser consumida por outras aplicações. Para este exemplo publicamos o Web Service em um servidor IIS, conforme ilustrado em "Chau Chee Yang Technical Blog" (ver seção **Links**).

Aplicação cliente

A partir de um Servidor de Aplicação, é possível atender aos mais variados tipos de cliente, incluindo aplicações escritas em outras linguagens, como Java e C#, e até mesmo de outras plataformas, como aplicações para Web e Mobile. Em se tratando do exemplo proposto para este artigo serão desenvolvidas três aplicações clientes em ASP .NET usando o Delphi Prism, PHP usando o RADPHP XE e por fim uma aplicação cliente do tipo VCL Forms usando o Delphi XE.

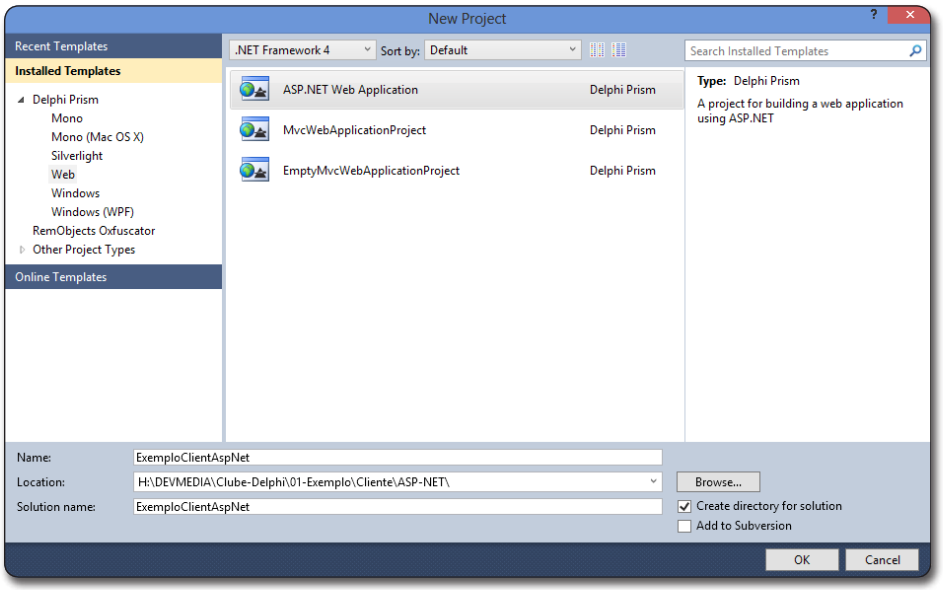

Figura 10. Wizard aplicação ASP.NET em Delphi Prism

Aplicação cliente em ASP.NET

Para esta aplicação cliente será necessário criar no Delphi Prism uma nova aplicação Web ASP.NET, conforme mostra a **Figura 10.**

Ao selecionar o item Web em Delphi Prism é carregada uma lista com possíveis projetos Web que podem ser desenvolvidos com Prism. No ComboBox que exibe .NET Framework 4 também é possível escolher qual versão do .NET será utilizada pela aplicação. No caso, foi selecionado a versão 4, e a solução foi salva a dentro da pasta "Cliente/ASP.NET" (**Figura 2).** Esta aplicação será bem simples, nela teremos apenas um botão para invocar a chamada ao Web Service e escrever o PDF no browser. Assim sendo, arraste um componente button para o design da página ASP.NET (**Figura 11**).

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DEVMEDIA

Antes de codificarmos o botão precisamos importar o caminho de referência para o Web Service. Com um clique do botão direito do mouse sobre a aplicação selecione a opção "Add Service Reference" que irá exibir um wizard para adicionar o serviço, conforme mostra a **Figura 12**.

Figura 11. Layout da aplicação Web em ASP.NET

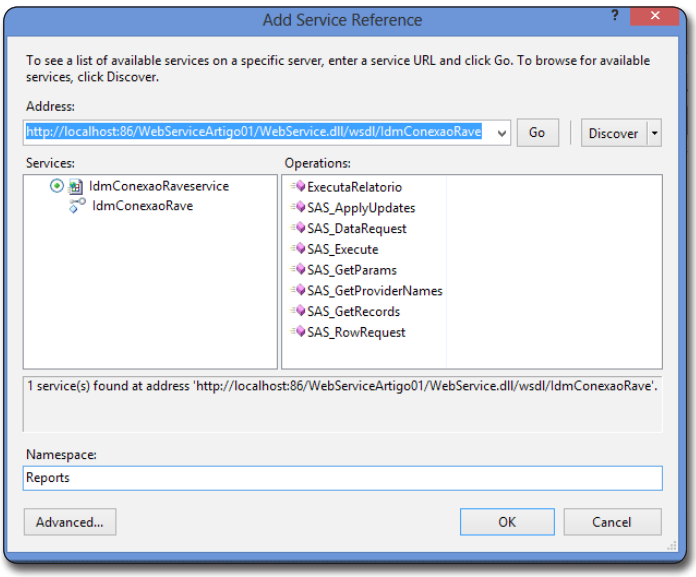

Figura 12. Wizard para adicionar uma referência web ao web service

O caminho será obtido ao executarmos o Web Service, e poderá ser copiado na URL do Web Browser que por default será http://localhost:<port>/<diretorio-virtual-criado-no-IIS>/WebService.dll/wsdl/IdmConexaoRave .

Digite o endereço na URL no wizard e utilize o botão GO, altere o namespace e clique no botão OK. Ao importar o arquivo observe que no Solution Explorer surgirá um novo item com as referências externas adicionadas ao projeto **(Figura 13).** Ao arquivo Web.config deve ser adicionado a declaração ao Web Service que encontra-se no app.config.

Adicionada a referência ao Web Service, podemos agora realizar a chamada da função embutida no Web Service ao evento onClick do botão, conforme demonstrado **Listagem 4,** a qual, se encontra comentada de acordo com a ação de cada bloco de código.

Aplicação cliente em PHP

Para esta aplicação será necessário criar no RadPHP XE uma nova aplicação do tipo Aplicação RCPL (**Figura 14**) salvando-a dentro da pasta "Cliente/PHP" **(Figura 2).** Esta aplicação cliente ao ser invocada automaticamente irá chamar o Web Service mostrando já no browser o relatório em pdf.

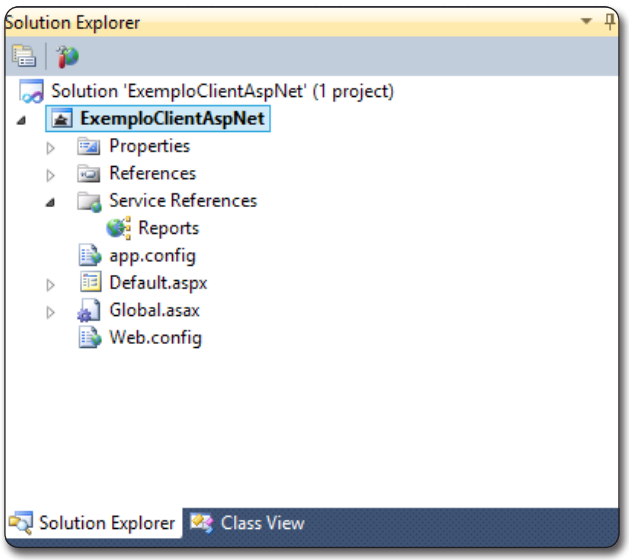

Figura 13. Referência ao Web Service adicionada ao projeto

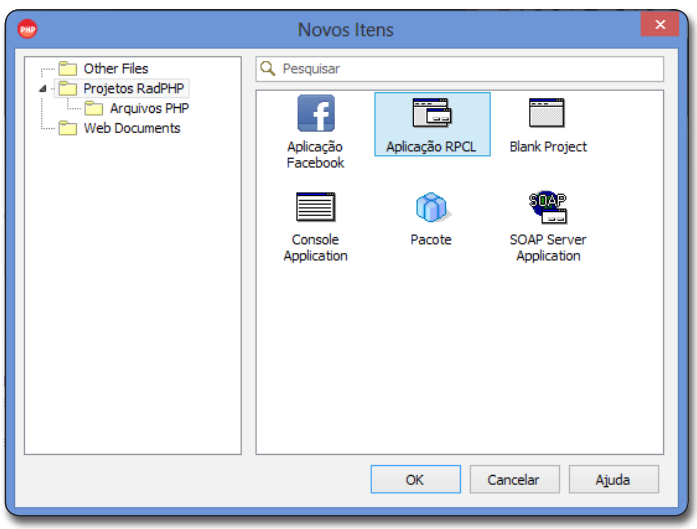

Figura 14. Wizard aplicação PHP em Rad PHP XE

Para que isso aconteça é necessário implementar o evento OnShow da página uni1.php conforme a **Listagem 5** cujo código encontra-se comentado de acordo com a ação de cada. Na **linha 06** da **Listagem 5** tem a criação de um objeto da classe MySoapClient conforme definida em "PHP Manual - Seção SOAP" (ver seção **Links**).

Aplicação cliente em VCL Forms

Para esta aplicação será necessário criar no Delphi XE uma nova aplicação do tipo VCL Forms Applications (**Figura 15**) salvando-a dentro da pasta "VCL-Forms" (**Figura 2).** Nesta aplicação colocamos dois buttons, um para gerar o relatório em pdf apenas acessando o método getAllAlunos() definido no servidor de aplicação DataSnap, o outro irá fazer uso do Web Service.

É necessário então para a primeira funcionalidade colocar o componente TSQLConnection e configurá-lo para acessar o servidor

de aplicação DataSnap. Com o acesso concluído é necessário gerar as DataSnap client classes, ou seja, a classe proxy de comunicação com o servidor de aplicação DataSnap. Os demais componentes e respectivas configurações são semelhantes aos usados para a criação do Web Service. Na **Listagem 6** apresentamos a implementação do button "Relatório" cujo código encontra-se comentado de acordo com a ação de cada bloco.

Já para a segunda funcionalidade é necessário utilizar o utilitário WSDL Importer que faz o mapeamento da WSDL e gera uma Unit com os métodos prontos para consumir os serviços, o que chamamos também de classe proxy.

Listagem 4. Implementação do método onClick do botão

```
01  method _Default.Button1_Click(sender: System.Object; e: System.EventArgs);
02  var
03  {declarando variáveis para criaçao do relatório}
04  relatorio : ExemploClientAspNet.Reports.IdmConexaoRaveClient;
05  retorno : boolean;
06  path : string;
07  fileName: string;
08  Buffer: array of Byte;
09  FileSize: Integer;
10  strReport: string;

11  {declarar uses System.IO}
12  MyFileStream: FileStream;
13  begin
14  //carga do relatório em pdf.
15  //cria o objeto de referencia ao WS
16  relatorio := new
    ExemploClientAspNet.Reports.IdmConexaoRaveClient();

17  //Define diretório e nome do arquivo
18  path := 'H:\DEVMEDIA\Clube-Delphi\01-Exemplo\bin\';
19  fileName  := 'getAllAlunos.pdf';

20  //faz a chamada ao WS. Aqui é criado o PDF
21  retorno := relatorio.ExecutaRelatorio;
22  strReport := path + fileName;

23  //Cria nova rotina de stream carregando o PDF
24  MyFileStream := FileStream.Create(strReport, FileMode.Open, FileAccess.Read);
25  strReport.Replace('\','');

26  //faz a leitura do tamanho do PDF
27  FileSize := MyFileStream.Length;
28  Array.Resize(Buffer,FileSize);// SetLength(Buffer, FileSize);
29  MyFileStream.Read(Buffer, 0, FileSize - 1);
30  MyFileStream.Close;
31  Response.BufferOutput := True;
32  Response.ClearContent;
33  Response.ClearHeaders;

34  //cria um novo cabeçalho na página do tipo PDF
35  Response.AddHeader('Content-Length', FileSize.ToString);

36  //escreve o arquivo PDF no browser
37  Response.AddHeader('content-disposition',  'inline;filename=' + strReport);
38  Response.ContentType := 'application/pdf';
39  Response.BinaryWrite(Buffer);
40  Response.Flush;
41  Response.Clear;
42  end;
```

Listagem 5. Implementação do método OnShow da unit1.php

```
01 function Page1Show($sender, $params)
02 {
03  //definição da wsdl do webservice
04  $wsdl_url = 'http://localhost:86/WebServiceArtigo01/WebService.dll/wsdl/
    IdmConexaoRave';

05  //criação do objetvo da classe MySoapClient
06  $cliente = new MySoapClient($wsdl_url);

07  //Define o diretório e o nome do arquivo
08  $path = 'H:\\DEVMEDIA\\Clube-Delphi\\01-Exemplo\\bin\\';
09  $fileName = 'getAllAlunos.pdf';

10  //chamada do método para executar o relatório
11  $retorno = $cliente->__call('ExecutaRelatorio',array());

12  $strReport = $path . $fileName;

13  //escreve o arquivo PDF no browser
14  $file = $strReport;
15  /* Note: Always use .pdf at the end.*/
16  header('Content-type: application/pdf');
17  header('Content-Disposition: inline;filename="' . $strReport . '"');
18  #header('Content-Transfer-Encoding: binary');
19  header('Content-Length: ' . filesize($file));
20  #header('Accept-Ranges: bytes');
21  @readfile($file);
22 }
```

Listagem 6. Implementação do método OnClick do Button "Relatorio"

```
01 try
02  //cria a conexão com a classe cliente
03  serverClasses := TServerMethods1Client.Create
    (SERVERCONEXAO.DBXConnection);

04  //busca os alunos
05  alunos-mestres;
06  dspRelAlunos.DataSet := serverClasses.getAllAlunos;
07  cdsRelAlunos.Open;

08  //chamar o relatório projeto do rave
09  RvRelatorio.ProjectFile := 'prsRelAlunos.rav';

10  //Define o path do PDF para exportação
11  RvSRelatorio.DoNativeOutput:= false;
12  RvSRelatorio.RenderObject := RvRenderPDF;
13  RvSRelatorio.OutputFileName:= 'getAllAlunos.pdf';

14  //Executa o relatório e faz com ele seja exportado
15  RvRelatorio.Execute;

16 finally
17  serverClasses.Free;
18 end;
```

Assim sendo, adicione um novo item ao projeto da categoria WebServices escolhendo o WSDL Importer conforme mostrado na **Figura 16**. Informe o endereço do descritor de serviço, no nosso caso o endereço da WSDL que foi indicada anteriormente **(Figura 12)**, salve a Unit gerada. Foi criada a interface de acesso ao Web Service e nesse momento é necessário colocar e configurar no Form principal da aplicação o componente THTTPPRIO que encontra-se na paleta de componentes WebService, configurando WSDLLocation, Port e Service que encontram-se comentados na Unit da Interface de acesso ao Web Service.

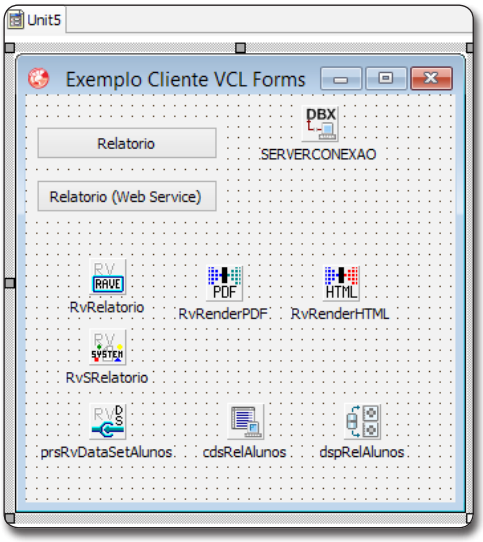

Figura 15. Layout aplicação VCL Forms

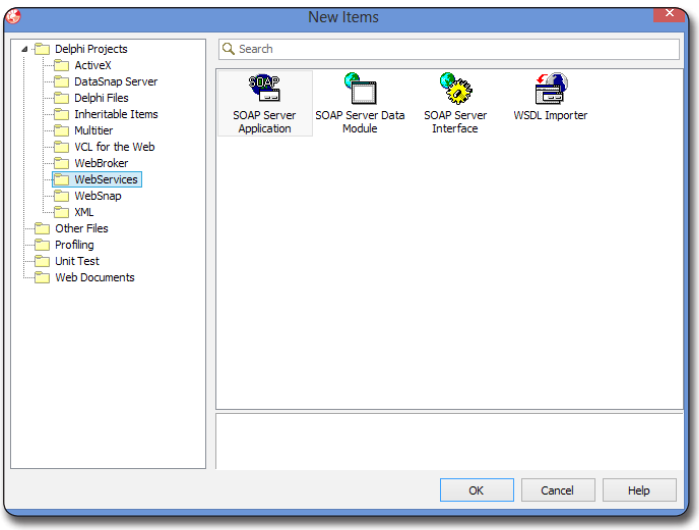

Figura 16. Wizard do WSDL Import

Finalmente, é possível implementar a ação do button "Relatório (WebService)" conforme **Listagem 7** que se encontra comentada de acordo com a ação de cada bloco de código. Por fim é só usar um dos componentes disponíveis em Delphi para a visualização de arquivos pdf e fazer a implementação.

Conclusão

Foi visto no decorrer deste artigo uma arquitetura de aplicação mista usando o framework DataSnap e Web Services para sanar a dificuldade dos desenvolvedores em lidar com relatórios em aplicações distribuídas. Assim sendo, mostrou-se que é possível utilizar os recursos tanto do framework DataSnap quanto do Web Service para a criação de relatórios e exibi-los em aplicações clientes de diferentes naturezas.

Por mais que sejam criadas três aplicações: servidor de aplicação DataSnap, Web Service e a aplicação cliente, em muitos casos a utilização desta arquitetura acaba trazendo benefícios em migrações

Listagem 7. Implementação do método OnClick do Button "Relatorio (WebService)"

```
01 try
02   //definindo o webservice
03   webSrv := HTTPRIO1 as IdmConexaoRave;

04   //caminho que será gerado o pdf
05   path := 'H:\\DEVMEDIA\\Clube-Delphi\\01-Exemplo\\bin\\';
06   fileName := 'getAllAlunos.pdf';

07   //chamada do método para executar o relatório
08   retorno := webSrv.ExecutaRelatorio;

09   strReport := path + fileName;

10   //usar qualquer compomente em delphi para mostrar neste ponto o
11   //arquivo pdf gerado e salvo no caminho de acordo com "strReport"

12 finally
13 end;
```

de sistemas implementados em Client/Server, onde os relatórios já estão praticamente todos desenvolvidos e o trabalho consiste em integrá-los a esta nova arquitetura.

Outra vantagem é a distribuição da aplicação, pois, a arquitetura mostrada possibilita a hospedagem do Banco de Dados, Servidor de Aplicação DataSnap, Web Service e Aplicação Cliente sejam hospedados em servidores distintos, garantindo a performance individual a cada uma delas além dos vantagens de uso de aplicações distribuídas.

Autor

Leandra Mendes do Vale
leandra@lsinformatica.com.br; leandramvale@gmail.com

Graduada em Ciência da Computação (UFU-1996), mestre em Redes de Computadores (UFU-2003), pós-graduada em Desenvolvimento JAVA (UNITRI-2004), pós-graduada em Educação à Distância pela Universidade Católica de Brasília (UCB-2008). Atualmente é professor titular da Faculdade Presidente Antonio Carlos de Araguari - UNIPAC e Coordenadora do Curso de Sistemas de Informação.

Links:

Entendendo os Padrões de Descrição de Web Services
http://www.devmedia.com.br/entendendo-os-padroes-de-descricao-de-web-services/28678

PHP Manual - Seção SOAP
http://br.php.net/manual/pt_BR/book.soap.php

Chau Chee Yang Technical Blog
http://chee-yang.blogspot.com.br/2009/10/configure-windows-7-iis7-for-isapi-dll.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Introdução ao UniDAC – Parte 1

Mais uma opção de componentes de acesso a dados no Delphi

Este artigo faz parte de um curso

Em se tratando de acesso a dados, mais recentemente, tanto a comunidade quanto o próprio Delphi têm convivido com um novo termo que está associado ao cenário de desenvolvimento de aplicações de banco de dados. O termo em questão é o sufixo "DAC", em reflexo ao AnyDAC, biblioteca de componentes de acesso a dados de terceiro adquirida há pouco tempo pela Embarcadero, e ao FireDAC, que foi o novo nome dado à esta mesma biblioteca no momento em que foi incorporada oficialmente ao Delphi. Sendo assim, pegando este gancho, o presente artigo expõe o UniDAC que, assim como as demais, também se refere a uma biblioteca de componentes de acesso a dados de terceiro. A diferença fundamental é que o UniDAC trata-se de uma alternativa paga (comercial) às opções nativas, cujo uso pode ser justificado pelos recursos que essa biblioteca dispõe, os quais serão vistos a seguir. Na seção **Links** encontra-se o acesso para sua versão trial.

Este tema será dividido em dois artigos que terão como foco a apresentação geral da biblioteca. Nesta primeira parte serão mostradas as características gerais do produto, bem como um overview de seus componentes.

Características gerais

O nome UniDAC, em linhas gerais, é um acrônimo para Universal Data Access Components que, numa tradução livre para o português, seria algo como "componentes de acesso a dados universal". Diz-se universal pelo fato dele prover acesso a diversos bancos de dados, atendendo a diversas ferramentas (Delphi, C++ Builder, Lazarus e Free Pascal) em diferentes plataformas (Windows, Mac OS, iOS, Linux e FreeBSD).

Resumo DevMan

Porque esse artigo é útil:

O tema é útil no desenvolvimento de qualquer aplicação que faça acesso a banco de dados. Através deste tipo de suíte é possível escolher dentre uma diversa gama de componentes já existentes, uma que se adapte melhor à sua necessidade, sempre considerando questões de performance, facilidade de uso, conectividade com o maior número de bancos de dados entre outros.

Este artigo tem como foco a apresentação do UniDAC, uma biblioteca de componentes de acesso a dados para Delphi produzida pela empresa Devart. Sendo assim, esta suíte comercial vem aumentar ainda mais o leque de opções para este contexto, onde bibliotecas como dbExpress e, mais recentemente, FireDAC, já são conhecidas do grande público. A semelhança entre todas elas fica por conta do suporte a diversos bancos de dados distintos, o que acaba por envolver os principais SGBDs do mercado, tais como Firebird, InterBase, Oracle, MySQL e Microsoft SQL Server.

Em se tratando do contexto do Delphi, o UniDAC é oficialmente compatível desde a versão 6 da ferramenta. Além disso, por questões óbvias, as plataformas atendidas se resumem necessariamente em Windows (32-bit e 64-bit), Mac OS X e iOS, conforme as especificações de suporte do próprio IDE. Sendo assim, o suporte completo às três plataformas citadas se dará somente sobre as versões XE2 e XE4 da ferramenta (vale lembrar que no XE3, o suporte a iOS foi retirado do Delphi).

Novamente falando em linhas gerais, em se tratando de acesso a dados, que é sua finalidade principal, o UniDAC provê suporte aos seguintes servidores de banco de dados:

- Oracle;
- Microsoft SQL Server;
- MySQL;
- PostgreSQL;
- InterBase;
- Firebird;

- SQLite;
- DB2;
- Microsoft Access;
- Sybase ADS (Advantage Database Server);
- Sybase ASE (Adaptive Server Enterprise);
- DBF;
- NexusDB.

Olhando pelo aspecto de estrutura, o UniDAC é composto essencialmente por dois elementos. O primeiro deles representa sua engine, ou seja, seu motor, que provê ao desenvolvedor uma interface de programação comum e unificada, receptível aos diversos bancos suportados. Já o segundo elemento, pode ser considerado sua parte mais fundamental, que é a sua camada de acesso a dados propriamente dita. Aqui ela é composta pelos provedores (providers) de acesso a dados, que irão proporcionar a interação entre a engine e os diversos servidores de banco de dados suportados. Cada provider fica então responsável por trabalhar com um servidor de banco de dados específico (Ex: TOracleUniProvider para Oracle, TInterBaseUniProvider para InterBase, TPostgreSQLUniProvider para PostgreSQL e assim por diante).

Em complemento, a **Figura 1** a seguir, retirada da própria página oficial do produto, ilustra todo esse suporte previsto bem como sua própria estrutura.

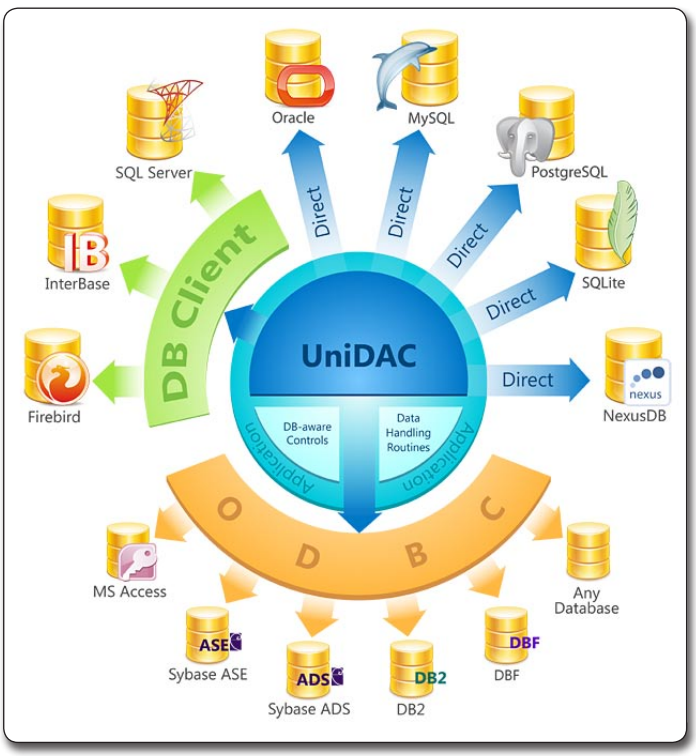

Figura 1. Visão geral do suporte previsto pelo UniDAC

Sob a visão do lado prático, alguns dos recursos peculiares do UniDAC, que podem eventualmente justificar sua escolha e utilização, são apresentados a seguir:

Conectividade

O UniDAC permite o trabalho com dados no modelo desconectado, tal qual o realizado nativamente com componente TClientDataSet, dispondo para isso de um controle automático de conexões, essencial para atividades com dados off-line. Adicionalmente, provê também o gerenciamento de tempo limite de conexão (Connection Timeout) para bancos Oracle, Microsoft SQL Server, MySQL e PostgreSQL.

Compatibilidade

O UniDAC, como biblioteca, acompanha o mercado de perto, a fim de estar preparado para trabalhar com as mais recentes versões dos servidores de banco de dados os quais suporta. Citando o Firebird, que é um dos SGBDs mais tradicionais em meio a comunidade Delphi, o provider do UniDAC é habilitado a trabalhar com sua versão estável (até o momento da escrita deste artigo) mais recente, a 2.5.x. Outro exemplo é o InterBase, SGBD da própria Embarcadero, cuja sua última versão (XE3) também é plenamente suportada por seu respectivo provider UniDAC.

Adicionalmente, a biblioteca provê a compatibilidade com os principais componentes para relatórios do Delphi, tal como o FastReport, trazido de forma nativa pelo IDE a partir da versão XE2. Da mesma forma, permite o trabalho com todos os controles visuais com acesso a banco de dados (Data-Aware), tanto da VCL quanto de terceiros.

Performance

Em vista de sua inevitável concorrência com as opções nativas e de mercado, o UniDAC prima por apresentar aspectos e recursos que lhe promovam destaque em meio aos demais. Obviamente, todos os seus pontos são construídos com essa intenção, no entanto, sua performance talvez seja um ponto de maior de atenção. Sendo assim, o UniDAC é tido como "rápido", proporcionando um alto desempenho em todos os seus processos que, fundamentalmente envolvem dados. Outros de seus recursos relacionados incluem pool de conexões (connection pooling), rápida localização de dados em objetos DataSet classificados/ordenados (sorted) e cache de campos do tipo Calculated e Lookup.

Usabilidade

Em termos de usabilidade, o UniDAC tem seu direcionamento principal voltado ao acesso direto aos dados do servidor dos principais SGBDs do mercado. Para estes casos, essa característica faz com que não haja a necessidade de instalação de outras camadas adicionais relacionadas a dados (Ex: ODBC – **BOX 1**). Além disso, para bancos como Oracle, MySQL e PostgreSQL, seus elementos de conexão proveem acesso sem o uso de bibliotecas cliente. Para o Delphi, a biblioteca possui versões tanto para a VCL quanto para FireMonkey (FMX).

InterBase e Firebird

Em se tratando do contexto Delphi, é possível afirmar que os dois SGBDs mais populares em meio à comunidade são InterBase

e Firebird. O primeiro, em razão de ser produzido pela mesma fabricante do IDE e, assim, acompanhar o produto em muitas das versões. Já o segundo dispensa comentários, uma vez que está presente em grande parte de exemplos e sistemas comerciais do cenário. Em função disso, vale ressaltar alguns aspectos do UniDAC, exclusivos a esses SGBDs:

- Suporte ao comando EXECUTE BLOCK e à cláusula RETURNING do Firebird;
- Suporte avançado a campos Blob;
- Suporte avançado a objetos Generator/Sequence;
- Componente específico (TUniTransaction) para trabalhar com transações, incluindo múltiplas transações;
- Componente de alerta (TUniAlerter) para trabalhar com eventos de bancos InterBase e Firebird.

BOX 1. ODBC

ODBC é o acrônimo para o termo em inglês Open Database Connectivity. De forma sucinta, pode ser definido como um padrão para acesso a SGBDs (Sistemas Gerenciadores de Banco de Dados). Sua implementação envolve uma especificação da linguagem SQL, o que provê um canal transparente e de fácil comunicação entre aplicação e base de dados. Devido seu padrão estrutural, seu uso está condicionado à existência de drivers específicos a cada SGBD, conhecidos no meio como drivers ODBC.

Edições

Comercialmente falando, o UniDAC apresenta duas edições, Standard e Professional, que se diferem basicamente entre três pontos fundamentais: características, recursos oferecidos e custo de aquisição.

Standard Edition

A edição Standard do UniDAC é tida como a mais simples, sendo disponibilizada como uma solução econômica aos desenvolvedores de aplicações de banco de dados. Em razão disso, o produto não contempla nenhum provider de dados, apenas sua engine padrão. Dessa forma, fica a cargo do desenvolvedor a aquisição dos providers que lhe forem cabíveis, mediante os bancos de dados utilizados em seu cenário. Uma limitação desta edição se dá justamente por seu suporte a banco de dados, que fica restrito a Oracle, Microsoft SQL Server, MySQL, InterBase/Firebird, PostgreSQL e SQLite. Por conseguinte, cada banco deste necessitará de seu respectivo provedor de dados, fornecidos pelo DAC (Data Access Components), vide ODAC, SDAC, MyDAC, IBDAC, PgDAC e LiteDAC, respectivamente.

Professional Edition

Em complemento à edição Standard está a Professional, que pode ser considerada a versão "completa" do produto. Isto porque, ela contempla todos os recursos e características da Standard Edition, com o adicional do suporte a todos os bancos de dados previstos (já citados anteriormente), bem como a inclusão de seus respectivos providers (**Figura 2**). Por fim, esta edição conta ainda com uma ferramenta de design-time exclusiva, denominada DataSet Manager, voltada ao gerenciamento de DataSets da aplicação.

Principais componentes

Uma vez que é instalado, o UniDAC disponibiliza os elementos de sua biblioteca na Tool Palette do IDE do Delphi, por meio de uma nova guia específica de componentes. Esta guia, que leva o mesmo nome do produto (**Figura 3**), apresenta então uma porção de componentes, cujos principais serão descritos a seguir. Inevitavelmente, muitos deles se tornam autoexplicativos, seja pela própria nomenclatura ou mesmo por sua proximidade com elementos similares nativos.

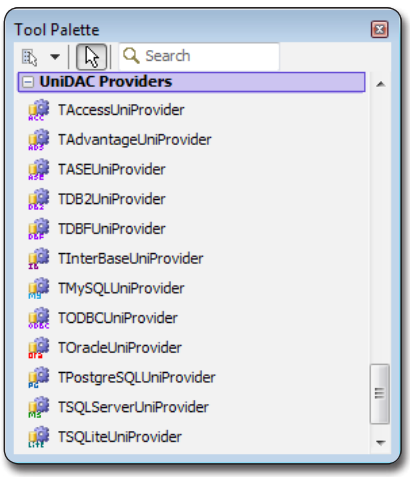

Figura 2. Providers UniDAC

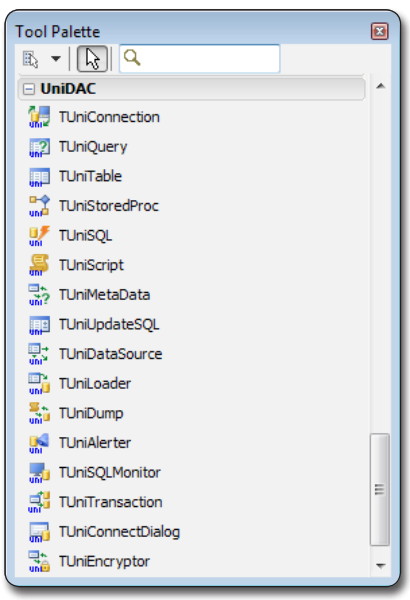

Figura 3. Tool Palette - guia UniDAC

TUniConnection

Este é o componente de conectividade da biblioteca. Isso significa dizer que é através dele que se dará a criação e controle da conexão com os servidores de banco de dados suportados. Dessa forma, os

demais componentes da biblioteca, quase que em sua totalidade (Ex: TUniQuery, TUniSQL e TUniScript), por estarem voltados a acesso a dados, necessitam essencialmente de um TUniConnection ativo, ligado à sua propriedade Connection, para operar. Sendo assim, é nele onde deverão ser passadas as informações de configuração da conexão que está se estabelecendo, tais como o servidor (Server), a porta (Port), o banco (Database) ao qual se quer acessar, além de suas credenciais de acesso (Username e Password). Adicionalmente, como peça-chave, também está incluso nessas informações o nome do provider de dados que será usado, relativo ao SGBD em questão. Mediante a escolha de determinado provedor de dados do UniDAC, novas opções de conexão podem ser especificadas, variando de provedor para provedor.

TUniQuery

Assim como os componentes Query providos nativamente pelo Delphi (Ex: TSQLQuery do dbExpress), o TUniQuery é o componente da biblioteca do UniDAC responsável pelas operações sobre os dados armazenados no banco, tais como a execução de consultas e atualizações. Uma das formas mais usuais de sua utilização se dá pela associação a um conjunto de registros, oriundos de uma tabela do banco e dados ou de uma seleção de dados (select). Para estes casos, de forma prática, o componente provê propriedades relacionadas às instruções SQL para cada operação a ser executada no servidor de banco de dados como, por exemplo, SQLInsert (para inserção), SQLDelete (para exclusão) e SQLUpdate (para atualização), tal como mostrado em destaque na **Figura 4**. Como diferencial, o TUniQuery apresenta características como o bloqueio automático de registros, que é então verificado antes de cada edição, além de efetuar um autorefresh depois de cada Post, o que faz com que os dados se mantenham sempre sincronizados (entre servidor e cliente).

TUniStoredProc

TUniStoredProc é o componente voltado ao acesso a procedimentos armazenados (objetos do tipo Stored Procedure) em bancos de dados. Sua utilização é tão simples tal qual o uso de elementos nativos já tradicionais, como TSQLStoredProc, do dbExpress. Isso significa dizer que sua principal ação se dá sobre sua propriedade StoredProcName, onde deverá ser informado o nome do objeto Stored Procedure que será acessado. A fim de facilitar este processo, a propriedade já lista todos os objetos deste tipo que estão disponíveis no servidor da conexão especificada em sua propriedade Connection. Uma vez feito isso, a instrução SQL para a chamada deste procedimento armazenado no servidor de banco de dados é gerada automaticamente pelo próprio componente, ficando transparente ao desenvolvedor. Num exemplo simples, o uso de seu método Execute em runtime, faz com que a Stored Procedure seja efetivamente executada no servidor.

TUniTable

Em teoria, o componente TUniTable tem seu comportamento muito aproximado ao do TUniQuery, uma vez que se fundamenta em recuperar e atualizar dados de um banco de dados. Todavia, em termos práticos, ao contrário do TUniQuery, o TUniTable não permite o trabalho com um conjunto de dados qualquer. Sendo assim, seu trabalho se dá exclusivamente com os dados de uma única tabela simples, inviabilizando, por exemplo, a efetivação de junções (Joins) com outras tabelas. Pelo fato de ser restrita a uma única tabela, a manipulação do componente não envolve a escrita de instruções SQL. Ao invés disso, basta que se especifique a tabela a qual se quer operar, informando seu nome na propriedade TableName. Novamente traçando um comparativo com as opções nativas, todo este comportamento do TUniTable é muito semelhante ao TSQLTable do dbExpress. Como complemento, exclusivamente para o seu processo de atualização de dados, o componente provê uma propriedade KeyFields, onde deverão ser informados os nomes dos campos "chave" a serem utilizados na montagem das instruções SQL de Update.

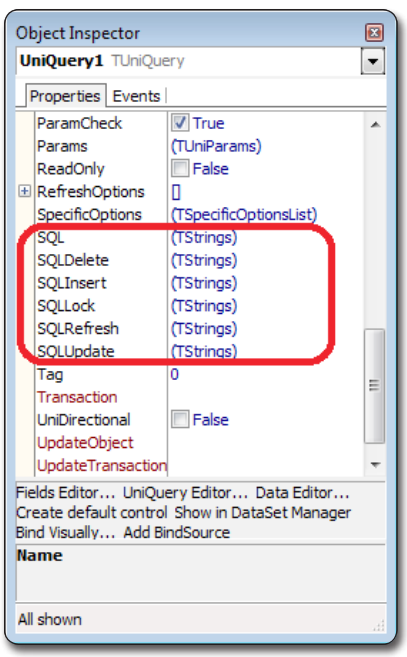

Figura 4. TUniQuery – Propriedades

TUniTransaction

Componente para controle transacional. No contexto do UniDAC, a conexão é provida pelo componente TUniConnection. Por este fato, a transação (TUniTransaction) se liga a uma conexão existente através de sua propriedade Connection. Dentre seus métodos disponíveis, estão alguns dos já tradicionais e conhecidos utilizados em diversos componentes transacionais semelhantes, tais como StartTransaction, Commit e Rollback. Para fins de conhecimento, o primeiro método inicia uma nova transação, o segundo "confirma" uma transação (persistindo de fato as alterações), enquanto que o terceiro descarta toda e qualquer modificação realizada sobre os dados no contexto da transação atual, encerrando-a por fim.

TUniSQL

De forma simples, o componente TUniSQL tem como função básica realizar operações diretas sobre um servidor de banco de dados. Tais operações podem ser resumidas em três principais: execução de instruções SQL, execução de blocos PL/SQL, bem como chamadas a Stored Procedures. O detalhe aqui fica por conta do componente apenas executar determinada instrução no banco de dados, excluindo assim operações que envolvam a recuperação de dados. A guia InterBase, presente na Tool Palette do Delphi, apresenta um componente similar ao TUniSQL, denominado TIBSQL (**Figura 5**).

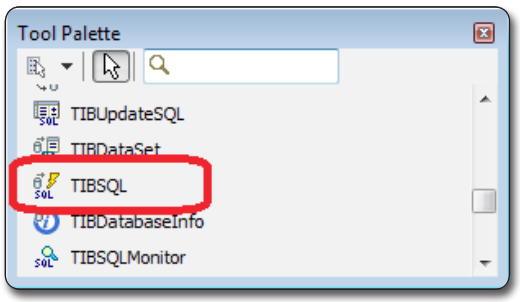

Figura 5. TIBSQL

TUniDataSource

Conforme sua própria nomenclatura sugere, o TUniDataSource é o componente da biblioteca UniDAC que se assemelha ao tradicional TDataSource do Delphi. Em termos técnicos, a classe TUniDataSource herda diretamente da classe TDataSource, obtendo assim todas as suas funcionalidades. Sendo assim, sua função principal é estabelecer uma "ligação" entre um elemento de dados (DataSet) do UniDAC e os controles visuais de dados presentes nos formulários da aplicação. No contexto Delphi, tais controles são tratados como "controles data-aware". Um adicional provido pelo componente é sua "linkagem" automática quando colocado próximo a um elemento DataSet compatível.

TUniSQLMonitor

Assim como o dbExpress tem seu componente de monitoramento das instruções SQL (TSQLMonitor) que estão sendo trafegadas entre aplicação cliente e servidor de banco de dados, o UniDAC também tem o seu. Tal elemento é denominado TUniSQLMonitor e seu objetivo é exatamente o mesmo. No aspecto prático, uma observação importante a se fazer é que cada aplicação Delphi que se utilize do UniDAC só poderá possuir apenas um único objeto TUniSQLMonitor ativo. Em termos técnicos, o componente apresenta apenas um único evento, denominado OnSQL, que é disparado no momento do rastreamento das atividades que envolvam SQL, dos componentes relacionados a banco de dados. A assinatura do método manipulador do evento OnSQL provê ao desenvolvedor, na forma de parâmetro (Text), a instrução SQL detectada no momento. Sendo assim, tal manipulador é o local ideal para a escrita da rotina necessária ao trabalho com as instruções SQL que estão sendo monitoradas. Num exemplo simples, tais valores poderiam ser direcionados a um arquivo de log da aplicação. Além disso, a ativação ou não do monitoramento efetuado pelo componente incide sobre sua propriedade booleana Active. Somente para fins demonstrativos, a **Figura 6** mostra os componentes SQLMonitor disponíveis nativamente (InterBase e dbExpress), além do TUniSQLMonitor.

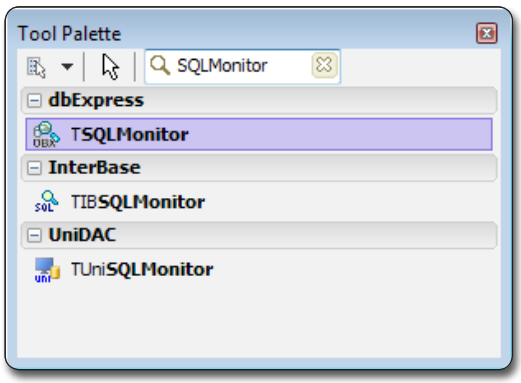

Figura 6. TSQLMonitor, TIBSQLMonitor e TUniSQLMonitor

Outros componentes

Além dos componentes tidos como tradicionais, tais como os abordados anteriormente, a biblioteca do UniDAC conta ainda com elementos mais peculiares, o que se torna de certa forma um atrativo do produto. A seguir são expostos alguns exemplos:

TUniAlerter

Dentre os elementos mais específicos do UniDAC está seu componente de alerta, o TUniAlerter. Numa definição mais formal, ele é apresentado como sendo o componente para envio e recebimento de eventos de banco de dados. Na prática, ele é utilizado para se disparar alertas na aplicação cliente, mediante as ações e eventos ocorridos no servidor de banco de dados. Todavia, para que todo este processo ocorra, é necessário que o SGBD em questão ofereça o devido suporte à notificação de eventos. Em razão disso, em vista dos SGBDs suportados pelo UniDAC, o componente TUniAlerter pode ser utilizado com quatro bancos diferentes, cada qual com seu recurso específico: InterBase (comando POST_EVENT), Firebird (comando POST_EVENT), Oracle (package DBMS_ALERT) e PostgreSQL (comandos NOTIFY e LISTEN).

Olhando para a sua estrutura, o componente apresenta alguns eventos e propriedades essenciais para o processo. Para citar alguns exemplos, OnEvent é chamado quando uma mensagem do servidor de banco de dados é enviada à aplicação, enquanto que seus métodos Start e Stop determinam o início e a parada do processo de esperar por novas mensagens.

TUniScript

Componente voltado à execução de instruções SQL. Seu diferencial fica por conta de permitir executar blocos com várias delas em sequência, uma a uma. No jargão do desenvolvimento, estas instruções SQL são chamadas de scripts. Sendo assim, o

componente acaba por exercer o papel fundamental de um executor de scripts. Em vista das muitas sentenças que um script possa conter, cada declaração necessita ser delimitada por caracteres ou palavras-chave, variando a cada SGBD. Em um exemplo para Oracle utiliza-se barra (/), enquanto que para SQL Server a palavra GO. Já em termos gerais, as declarações são separadas por ponto e vírgula (;), conforme exemplo mostrado na **Listagem 1**.

Listagem 1. Exemplo de script SQL para o banco Employee, que acompanha o InterBase

```
01 ALTER TABLE EMPLOYEE ADD PRIMARY KEY (EMP_NO);
02 ALTER TABLE EMPLOYEE ADD FOREIGN KEY (DEPT_NO) REFERENCES
   DEPARTMENT
   (DEPT_NO);

03 ALTER TABLE EMPLOYEE ADD FOREIGN KEY (JOB_CODE, JOB_GRADE,
   JOB_COUNTRY)
   REFERENCES JOB (JOB_CODE, JOB_GRADE, JOB_COUNTRY);

04 CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

Assim como ocorre com a grande maioria das ferramentas de scripts de banco de dados, o comportamento do TUniScript provê a execução contínua das instruções, mesmo em caso de erros. Na prática, isso significa dizer que, caso alguma delas apresente erro, a devida exceção é mostrada, mas o processo continua com a execução da próxima instrução e assim sucessivamente.

TUniMetaData

Conforme seu nome sugere, o componente TUniMetaData é um elemento voltado ao trabalho com as meta-informações (metadados) dos objetos do banco de dados, tais como tabelas, campos e triggers. Sendo assim, ele é capaz de obter tais informações do servidor, tornando-as disponíveis à aplicação cliente. Em termos de estrutura, dentre suas propriedades disponibilizadas, algumas merecem destaque, tais como Connection e Restrictions. A primeira, especifica a conexão provida por um TUniConnection, que será utilizada pelo componente para obter as informações de metadados dos objetos do servidor de banco de dados. Já a segunda, estabelece condições de restrição para a lista de objetos a serem descritos. Além destas, MetaDataKind é uma propriedade do tipo string que especifica o tipo de meta-informação a ser mostrada. Por padrão, todos os tipos possíveis já vêm habilitados, sendo representados por uma string vazia. A seguir, são listados os tipos disponíveis e a especificação de cada um:

- Databases: mostra as meta-informações do banco de dados;
- Tables: mostra as meta-informações das tabelas do banco;
- Columns: mostra as meta-informações das colunas das tabelas do banco;
- Procedures: mostra as meta-informações dos objetos Stored Procedure presentes no banco;
- ProceduresParameters: mostra as meta-informações dos parâmetros de objetos Stored Procedure presentes no banco;
- Constraints: mostra as meta-informações das eventuais constraints definidas no banco de dados;
- Indexes: mostra as meta-informações dos índices presentes no banco de dados;
- IndexColumns: mostra as meta-informações de colunas indexadas presentes nas tabelas do banco;
- MetaDataKinds: habilita todos os tipos disponíveis. Mesmo resultado obtido ao atribuir uma string vazia como valor para a propriedade.

TUniLoader

Permite a carga rápida de dados externos para uma tabela do banco de dados. De forma geral, o TUniLoader fornece através de suas propriedades, as configurações necessárias para a efetivação do processo. A primeira delas é a propriedade Connection, que receberá uma conexão válida com o banco de dados em questão. Em complemento, TableName recebe o nome da tabela para qual a carga será feita. Adicionalmente, a propriedade Columns pode ser utilizada para se acessar individualmente cada coluna da tabela. Dentre os métodos disponibilizados pelo componente, o método Load dá início ao processo de carga de dados à tabela.

Para cada tipo de SGBD, o componente adota uma estratégia diferente a fim de otimizar ao máximo o processo de carga. Num exemplo simples para bancos MySQL, a carga se baseia em instruções SQL de INSERT simples. Já para bancos Firebird 2.x, o componente faz a combinação destas com o uso da instrução EXECUTE BLOCK.

TVirtualTable

Até o momento, todos os componentes apresentados se referem aos presentes na guia UniDAC da paleta de componentes do IDE, todavia, ao instalar a biblioteca, dois deles são direcionados à guia Data Access. O primeiro destes é denominado TVirtualTable e é destinado ao armazenamento de dados em memória. A definição da estrutura da tabela virtual se dá basicamente de duas formas. A primeira envolve a definição de seus campos (Fields) de forma manual, que pode ser feita tanto em tempo de design, através do Fields Editor, ou em runtime, por meio de seu método AddField. Outra forma de sua estruturação pode ser feita através do carregamento de um arquivo de tabela existente, mediante o formato exigido pelo componente.

Em se tratando da definição manual de seus campos, o estado inicial da tabela virtual representada pelo componente é, obviamente, vazio (sem dados). Uma vez que os dados começam a ser lançados nesta tabela, permanecem persistidos até o momento do fechamento (Close) do componente. Após isso, todas as informações são descartadas, uma vez que estavam apenas alocadas em memória. Para manter os dados armazenados, o componente provê novamente duas possibilidades, em design-time e runtime. A primeira envolve o uso da opção voStored de sua propriedade Options. Já para a segunda, assim como um TClientDataSet nativo, o TVirtualTable também provê um método SaveToFile, para a persistência em um arquivo em disco de toda estrutura e dados envolvidos.

TCRBatchMove

TCRBatchMove é o outro componente UniDAC direcionado à guia Data Access da Tool Palette do IDE. Sua finalidade é a de realizar operações em lote envolvendo DataSets, mais especificamente a transferência de dados entre objetos deste tipo. Sendo assim, sua utilização básica envolve a cópia de um conjunto de registros de um DataSet para outro ou mesmo a exclusão dos registros de um DataSet correspondentes em outro. Os eventuais objetos a serem envolvidos no processo são então definidos através das propriedades do próprio TCRBatchMove, sendo Source para o "DataSet origem" e Destination para o "DataSet destino". A efetiva execução da operação se dá por meio da chamada ao seu método Execute.

Dentre as diversas propriedades disponibilizadas pelo componente, duas merecem destaque: MovedCount e ProblemCount. A primeira retorna a quantidade de registros lidos do DataSet origem durante a operação, já a segunda retorna a quantidade de registros que não puderam ser adicionados ao DataSet destino devido a incompatibilidade de tipo de campo.

TUniConnectDialog

Conforme o próprio nome sugere, TUniConnectDialog trata-se de um elemento que provê ao usuário final da aplicação uma caixa de diálogo relacionada à conexão de banco de dados. Neste diálogo o usuário deverá então informar os dados básicos de acesso à mesma, tais como o nome do provider de dados, o nome do servidor de banco de dados, o nome do banco ao qual se quer acessar, o nome do usuário de acesso, senha e número da porta. O componente provê ainda, através de suas próprias propriedades (**Figura 7**), uma total customização da aparência visual da caixa de diálogo.

A seguir, segue alguns exemplos destas:

- Caption: especifica o caption da janela;
- CancelButton, ConnectButton: cada propriedade desta especifica um label para os botões relacionados – Cancel e Connect, respectivamente;
- DatabaseLabel, PasswordLabel, PortLabel, ProviderLabel, ServerLabel e UsernameLabel: cada propriedade especifica um prompt para o edit relacionado – nome do banco, senha, número da porta, provider, servidor e nome do usuário, respectivamente.

Além das propriedades relacionadas ao seu visual, o TUniConnectDialog provê ainda algumas outras voltadas ao aspecto funcional. Num exemplo imediato, é possível citar as propriedades Retries, que indica o número de tentativas falhas de conexão, e StoreLogInfo, que uma vez marcada, faz com que as informações

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Clube Delphi

CONHEÇA ALGUNS DOS CURSOS:

- Curso de Multicamadas com Delphi e DataSnap
- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism

Para mais informações :
www.devmedia.com.br/cursos/delphi
(21) 3382-5038

DEVMEDIA

de acesso sejam persistidas no registro do sistema (para futuros acessos), uma vez que a conexão tenha sido estabelecida. Adicionalmente, a classe TUniConnectDialog contempla ainda dois métodos, Execute e GetServerList, oriundos de sua classe base TCustonConnectDialog. O primeiro, tido como principal, exerce a função de efetivamente mostrar o diálogo da conexão ao usuário, além de, internamente, invocar o método Connect da conexão, uma vez que o botão relacionado seja acionado em tela. Por fim, GetServerList retorna uma lista com os nomes dos servidores disponíveis na ocasião.

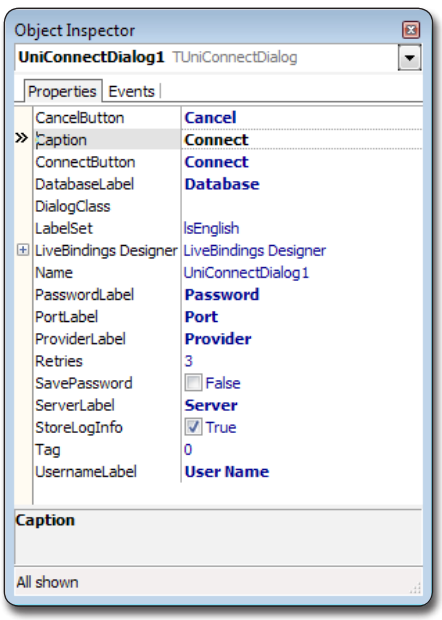

Figura 7. TUniConnectDialog – Propriedades

TUniDump

TUniDump é um componente com um comportamento bastante peculiar. Em termos gerais, sua especificação o designa a armazenar e restaurar dados de tabelas (ou Views editáveis) do banco de dados, através da utilização de scripts. Na prática, citando o armazenamento de dados, utiliza-se sua propriedade TableNames para se especificar a lista de objetos do banco que terão seus dados armazenados. Por conseguinte, a efetiva geração dos scripts relacionados se dá pelo método Backup.

Conclusão

A parte de acesso a dados no Delphi, em termos de tecnologia e componentes, sempre foi muito bem resolvida por meio de suas opções nativas. O exemplo mais forte para essa afirmação é o já conhecido dbExpress, que acompanha a ferramenta de longa data, se estabelecendo como uma das principais opções no contexto de banco de dados. Todavia, em determinados cenários, os recursos providos pelo próprio dbExpress podem não atender de forma eficaz a necessidade do projeto. Em situações como esta é que as opções de terceiro ganham força, apresentando características-chave para contextos específicos não atendidos pelas opções nativas. Em vista disso, no presente artigo pudemos conhecer de forma geral o UniDAC, que se trata de um produto comercial (pago) produzido pela empresa Devart. Mais do que um simples produto, ele se apresenta como sendo mais uma opção, dentre as várias disponíveis no mercado atual, de biblioteca de componentes de acesso a dados para Delphi.

Por conseguinte, obviamente há no mercado uma gama de outras bibliotecas, incluindo opções gratuitas, voltadas não só para acesso a dados, como também para os mais variados fins. Isso se confirma aqui mesmo nesta revista, onde muitas dessas bibliotecas já foram (e continuarão sendo) tema de artigos, vide AnyDAC, Jedi VCL, DevExpress, entre outras.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

Delphi XE4 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE4 – Página oficial do produto
http://www.embarcadero.com/products/delphi

UniDAC – Download Trial
http://www.devart.com/unidac/download.html

UniDAC - Página oficial do produto
http://www.devart.com/unidac

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Exposição de objetos de negócio em JSON por servidor REST

Conheça o framework mORMot

A arquitetura tem uma grande importância no ciclo de vida de um projeto de software, sendo a principal responsável pela forma como uma aplicação é organizada. Quanto maior o projeto, maior a sua importância. Não se trata de uma fase do desenvolvimento de software, a arquitetura está presente durante toda a vida do projeto e por isso precisa ser constantemente evoluída. Para isso ser possível, é importante que ela seja flexível e extensível, caso contrário, alterações em qualquer base do projeto gerarão uma grande dor de cabeça.

Se analisarmos alguns frameworks e APIs de grande utilização em projetos Delphi, é possível perceber que quem os desenvolveu não estava preocupado com a arquitetura dos softwares que usam essas bibliotecas, em alguns casos, a maior preocupação é ser RAD, ser "fácil" de utilizar, possibilitar a construção rápida de uma aplicação inteira através da interface do IDE, sem a necessidade de digitar qualquer código. Felizmente este cenário está mudando, ao menos, no que diz respeito aos frameworks Open Source para Delphi.

O fato é que grande parte dos projetos Delphi, às vezes por influência da plataforma como um todo, ou simplesmente por falta de opções, são projetos que foram desenvolvidos usando a metodologia RAD. São projetos onde é possível encontrar componentes de acesso a dados na interface gráfica, SQL em baixo de botões, e assim por diante.

Ainda na época em que o Delphi era mantido pela Borland, uma empresa chamada BoldSoft tentou mudar esse cenário, em busca de um apelo mais orientado a objetos e menos RAD, com a criação do projeto Bold, que era um ORM. Infelizmente, a Borland comprou a BoldSoft e matou o projeto. Mais tarde surgiu o ECO que também foi descontinuado.

Resumo DevMan

Porque esse artigo é útil:

Vamos abordar um framework chamado mORMot. Nele temos concentrados dois pilares para quem desenvolve um sistema corporativo. Temos um servidor de aplicações REST e um ORM (Object-Relational-Mapper). Esses também são pontos importantes quando se desenvolve soluções voltadas à arquitetura orientada a objetos para projetos multicamadas com SOA (Service Oriented Architecture). Do ponto de vista técnico, o servidor REST do mORMot permite que objetos trafeguem por seu aplicativo enquanto que permite persistir esses mesmos objetos em um banco de dados relacional, já do ponto de vista de negócios, possuir um servidor REST garante que qualquer cliente que implemente o protocolo HTTP pode consumir seu servidor, com total independência de tecnologia, e possuir um ORM à mão traz a independência quanto ao banco de dados.

Na busca por soluções para os problemas de arquitetura de software, uma das primeiras preocupações é a separação em camadas. Entre as camadas comumente utilizadas, está a camada de acesso a bancos de dados ou camada de persistência. Esta costuma apresentar os primeiros desafios para a arquitetura de software de projetos Delphi. Enquanto plataformas como Java e .NET possuem frameworks ORMs maduros, para Delphi temos poucas opções e todas são relativamente novas. Dentre as opções disponíveis, a maioria surgiu após o Delphi 2009 e a adição dos novos recursos à linguagem, como Generics e a nova RTTI. Como não há retro compatibilidade com versões antigas do Delphi, para poder usufruir desses frameworks é necessário migrar as aplicações, o que costuma trazer grandes desafios e nem sempre é viável.

Dentre as principais opções de framework ORM para Delphi, temos o DORM, TMS Aurelius e o mORMot. Tanto o DORM como o Aurelius são frameworks exclusivamente de mapeamento objeto

relacional, já o mORMot é mais do que isso, tendo o ORM como uma de suas principais funcionalidades.

Um ORM traz algumas vantagens interessantes, como eliminar em grande parte a necessidade de uso de SQL, diminuir o tempo de desenvolvimento e permitir que uma única estrutura de dados seja usada em mais de um banco de dados. Por outro lado, passam a surgir algumas preocupações, como o desempenho. Um framework ORM pode trazer uma perda de desempenho devido ao processamento que é realizado para a geração de SQL e mapeamento objeto relacional com a base de dados. Essa é uma preocupação que os desenvolvedores do mORMot tem desde o início do projeto e um diferencial do projeto.

O framework

O mORMot é um framework ORM e SOA (**BOX 1**) que traz consigo uma enorme gama de funcionalidades, como ferramentas de log, MVC, Unicode, internacionalização, cache, testes e etc. Compatível com as versões do Delphi 6 a XE 4 é um dos poucos frameworks que oferecem maior retro compatibilidade, o que é uma boa notícia para quem ainda usa versões antigas e não quer depender de uma migração para usufruir de ferramentas modernas.

BOX 1. SOA

SOA é um acrônimo para arquitetura orientada a serviço, mas pode-se também ser considerada uma visão de como aplicações heterogêneas deveriam ser desenvolvidas e integradas. O principal objetivo da SOA é saltar das aplicações monolíticas para um conjunto de serviços reutilizáveis que podem ser compostos para criar aplicações.

As principais funcionalidades do mORMot são o ORM e sua camada de serviços. O ORM traz um conceito diferente e inovador, o qual os autores chamam de ORM cliente-servidor. Isso significa que o framework trabalha em todas as camadas da aplicação. O provedor de serviços permite de uma forma muito simples que serviços sejam disponibilizados através de REST. Assim como o ORM, as funcionalidades de serviços também atuam em todas as camadas da aplicação, simplificando o trabalho do desenvolvedor.

Desenvolvido com pensamento voltado para um design simples (KISS – Keep It Simple Stupid), baseado em Domain Driven Design, o mORMot é um projeto ambicioso e inovador que tem como preocupações centrais a versatilidade e o desempenho. Sua arquitetura e implementação orientada a objetos fogem totalmente do ambiente RAD encontrado na plataforma Delphi, e esse é um dos seus principais diferenciais. Por maior que sejam as diferenças com os frameworks da Embarcadero, como o DataSnap, a adaptação não é um problema para quem conhece os conceitos básicos de OO.

Client-Server ORM / SOA

O mORMot implementa uma arquitetura cliente-servidor RESTFull (**BOX 2**) seguindo as melhores práticas de MVC, N-Tier, ORM e SOA.

BOX 2. RESTFull

REST (Representational State Transfer) é uma técnica de engenharia de software para sistemas hipermídia distribuídos como a World Wide Web. O termo se originou no ano de 2000, em uma tese de doutorado (PHD) sobre a web escrita por Roy Fielding, um dos principais autores da especificação do protocolo HTTP que é utilizado por sites da internet. Os sistemas que seguem os princípios REST geralmente são chamados de RESTful. Uma aplicação RESTFul combina o uso dos princípios estabelecidos pela técnica do REST que podemos destacar como os principais:

1 – Possui um protocolo cliente/servidor sem estado, ou seja, cada mensagem HTTP contém toda a informação necessária para compreender o pedido;

2 – Uso de um conjunto bem definido de operações que se aplica a todos os recursos de informações. As operações mais importantes são POST, GET, PUT e DELETE que podem ser comparadas com operações CRUD para persistência de dados;

3 – Cada recurso no sistema REST é unicamente direcionado através da sua URI - Uniform Resource Identifier.

REST é basicamente uma forma de pensar, não um protocolo ou padrão, com ele temos um estilo de projetar aplicativos fracamente acoplados que contam com recursos nomeados. Talvez a parte mais complicada em uma aplicação REST é definir quais recursos você vai expor e como serão os critérios de segurança.

Diferentes aplicativos clientes podem acessar o mesmo servidor remoto ou local, usando diferentes protocolos de comunicação, como HTTP, named pipes ou mensagens GDI. A **Figura 1** ilustra a arquitetura do mORMot.

Mudar um aplicativo com uma arquitetura Standalone para multicamadas é somente uma questão de organizar a forma como algumas classes do mORMot são inicializadas. Um mesmo executável pode "rodar" como Standalone, servidor ou cliente, dependendo de parametrizações que podem ser feitas em Runtime.

O ORM cliente-servidor significa que é possível utilizar o framework exatamente da mesma forma em todas as camadas da aplicação. Por exemplo, o comando "Basedados.

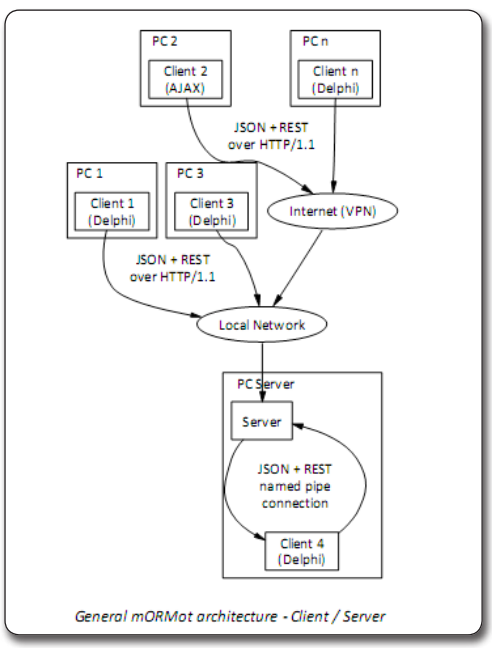

Figura 1. Arquitetura Cliente/Servidor do mORMot

Add(Pessoa)" é responsável por adicionar à base de dados um objeto do tipo pessoa, onde "Basedados" é uma referência ao ORM. Esse mesmo comando pode ser utilizado tanto no servidor como em camadas clientes. Quando for utilizado no servidor, o framework simplesmente fará o mapeamento objeto relacional e enviará as informações para a base de dados. No caso de ser executado em uma aplicação cliente, o framework fará uma requisição REST para o servidor, o qual realizará a gravação dos dados. Não é necessário se preocupar com a implementação da comunicação com o servidor, pois o framework é responsável por isso.

Principais vantagens

O mORMot possui muitos outros recursos, que o diferenciam dos demais, sendo alguns deles:

- Sistema de cache otimizado com atualização inteligente através da arquitetura RESTFull;
- SOA usando serviços RESTFull customizados com JSON. É possível trafegar entre as camadas qualquer variável primitiva ou instâncias de objetos dos tipos TStrings, TCollection, TPersistent ou TObject (através de um serializer customizado), assim como arrays dinâmicos e records;
- Sistema de autenticação RESTFull através de um sistema de segurança duplo (sessão e requisições);
- Biblioteca de criação e análise de JSON extremamente rápida;
- Servidor HTTP de alta performance usando http.sys;
- Usa SQLite3 como kernel mas permite a conexão com qualquer outro banco de dados (via OleDB / ODBC / Zeos ou bibliotecas de conexão direta, como Oracle);
- Não depende da unit DB.pas e de nenhum componente de terceiro, o que faz com que a versão Delphi Starter Edition seja suficiente para usar o framework;
- Possui classes para trabalhar, opcionalmente, em conjunto com soluções baseadas no DB.pas, como NexusDB, DBExpress, FireDAC, AnyDAC, uniDAC e até BDE;
- Permite usar SQL e requisições RESTFull com múltiplos bancos de dados ao mesmo tempo (graças ao mecanismo de tabelas virtuais do SQLite3);
- Sistema de pesquisa de texto com algoritmo otimizado semelhante ao do Google;
- Geração automática de interfaces gráficas. Grids são criados em Runtime juntamente com telas baseadas nos componentes Ribbon (Office). O código simplesmente define as ações e liga elas às tabelas para que toda a interface seja construída a partir de algumas linhas de código;
- Sistema de relatórios integrado que é capaz de gerar PDFs complexos;
- Desenhado para ser o mais rápido possível. Usa assembly quando necessário, buffers para diminuir consumo de memória e etc;
- Arquitetura multithread;
- Mais de mil páginas de documentação;
- Clientes Delphi e AJAX podem consumir o mesmo servidor;
- Biblioteca cliente em JavaScript com Ext.JS (ver sessão **Links** ao final deste artigo);
- Todo o código fonte está disponível;
- Funciona do Delphi 6 ao XE4;
- Suporte ao Win32 e Win64;
- Totalmente Unicode em qualquer versão do Delphi;
- Cobertura de testes automatizados (garantia contra regressão);

Unicode e UTF8

O mORMot possui 100% de compatibilidade com Unicode, trabalhando sempre com UTF8 mesmo para versões antigas do IDE (Delphi 6, por exemplo). Em busca de uma solução para o problema que foi criado com a alteração do tipo string para UnicodeString nas versões mais recentes, o framework possui alguns novos tipos de dados. Isso também facilita o suporte a todas as versões do Delphi sem a necessidade de diretivas de compilação para os tipos String. Também é útil quando temos camadas implementadas em versões diferentes. É possível implementar uma camada servidora em Delphi XE4 e cliente em Delphi 6, onde não será necessário se preocupar com o tipo string, bastando simplesmente usar os tipos Unicode do framework.

Um dos tipos que usaremos no exemplo é o RawUTF8. Este é o tipo mais recomendado para trabalhar com qualquer tipo de strings com o framework, embora não seja obrigatório. Nada impede de você trabalhar com o tipo String. Também estão disponíveis diversas funções otimizadas e multithread que ajudam com o tratamento de conversões e Unicode.

Infrastructure – Sistema de Logs

O mORMot possui várias classes de infraestrutura, dentre elas um sistema completo de log. Dentre as principais funcionalidades estão:

- Permite habilitar o log em diversos níveis;
- Carregar os símbolos (arquivo map) para uso nos logs (Ex: nome do arquivo fonte e número da linha do código);
- Compressão do arquivo .map para binário (.mab) (900KB -> 70KB);
- Inclusão do arquivo .map/.mab no executável, com um acréscimo muito pequeno no tamanho;
- Logs de exceções contendo nome da unit e número da linha;
- Stacktrace (como madExcept)
- Timestamps de alta precisão, para medir o desempenho da aplicação;
- Serialização de JSON para Sets, Enum, TList, Tpersistent, TObjectList e arrays dinâmicos;
- Log por Thread ou global;

Documentação e suporte

Ao avaliar uma solução Open Source é comum algumas preocupações com relação à documentação, suporte, correções de bugs e eventuais implementações de novas funcionalidades. Não é incomum encontrarmos projetos Open Source parados no tempo, com listas de discussão desertas e sem qualquer documentação,

e isso é um risco para qualquer projeto. Porém, esses itens são justamente o ponto mais forte do mORMot.

Mesmo comparado a soluções pagas, ele se destaca. A documentação disponível é extensa e detalhada (sessão **Links**), que é gerada através da própria documentação do código, ou seja, o código é muito bem documentado e atualizado.

Correções de bugs geralmente são liberadas em menos de 24 horas, o que é extremamente positivo se comparado ao que ocorre com bugs das bibliotecas da Embarcadero, por exemplo, onde é comum esperar pela próxima versão do Delphi para um hotfix do problema.

O fórum de discussão (sessão **Links**) funciona excepcionalmente bem. Administrado pelo o próprio patrocinador do projeto, geralmente oferece respostas rápidas e precisas. Além disso, o projeto é muito ativo, como mostra a timeline que exibe atividades diárias.

É incomum encontrar projetos tão ativos, bem documentados e suportados como este, e que ainda contam com uma grande cobertura de testes automatizados.

Instalação e configuração

Entre as opções de download do framework, pode-se baixar um arquivo zip contendo a última versão estável, o último build diário, ou clonar o repositório (trunk). Para quem pretende começar a utilizar o framework agora, é recomendado utilizar a versão mais recente (instável), já que houve muitas mudanças da versão 1.17 para a 1.18 (em desenvolvimento). Na seção **Links** encontra-se o acesso para download do projeto.

Depois de baixar e descompactar os arquivos do framework (mORMotNightlyBuild.zip) e do SQLite3 (sqlite3obj.7z), basta adicionar os diretórios ao Library Path do Delphi, disponível através do menu Tools>Options>Environment Options>Delphi Options>Library.

Juntamente com os projetos de exemplo está disponível uma pasta com uma versão do mORMot sem a documentação (por questão de tamanho), o que permitirá a qualquer um baixá-lo e testá-lo sem precisar se preocupar com a instalação e configuração do framework nesse primeiro momento.

Projeto exemplo - ORM com SQLite

O projeto de exemplo visa mostrar de forma bem simples o funcionamento do ORM não se aprofundando em todas as funcionalidades do framework. Sendo assim, ele consistirá em duas aplicações consumindo uma base de dados SQLite através do ORM do mORMot. Primeiramente criaremos o aplicativo que vai servir como uma camada servidora. Este fornecerá acesso à camada de persistência por meio de REST.

Para o servidor de aplicação criamos um novo projeto VCL sem a necessidade de qualquer componente, podendo ser uma aplicação console ou até mesmo um serviço. A **Figura 2** ilustra um simples executável com uma imagem, na qual representará nosso Server.

Dividimos a inicialização deste servidor em três partes, sendo elas: configuração do sistema de logs, inicialização do banco de dados e inicialização do servidor HTTP. Estas etapas serão representadas por três métodos, sendo eles: "ConfigurarLog", "InicializarBancodeDados" e "InicializarServidorHTTP".

O código da **Listagem 1** simplesmente ilustra a definição dos métodos citados e da cláusula uses, definidos no formulário da **Figura 2**.

Figura 2. Aplicativo servidor

Listagem 1. Formulário do servidor

```
01 ...
02 uses
03   mORMotSQLite3, mORMot, mORMotHttpServer;

04 TFrmServidor = class(TForm)
05   private
06     ModeloDados: TSQLModel;
07     BaseDados: TSQLRestServerDB;
08     ServidorHTTP: TSQLHttpServer;
09     procedure ConfigurarLog;
10     procedure InicializarBancoDeDados;
11     procedure InicializarServidorHTTP;
12   end;
```

Configuração do sistema de logs

Para o nosso exemplo usaremos um log que nos permitirá auditar todos os comandos SQL executados na base de dados pelo framework. A implementação é muito simples, como mostra a **Listagem 2**. Os logs serão gravados no diretório da aplicação.

Para maiores informações de como configurar e customizar o sistema de logs pode-se consultar a página 78 da documentação (na data deste artigo).

Listagem 2. Método ConfigurarLog

```
01 procedure TFrmServidor.ConfigurarLog;
02 begin
03   TSQLLog.Family.Level := LOG_VERBOSE;
04 end;
```

Inicialização do banco de dados

Para iniciarmos o banco de dados SQLite precisamos, primeiramente, definir qual será o modelo do banco. O modelo é representado no framework pela classe TSQLModel que basicamente é o conjunto de todos as classes/entidades que serão persistidas. Para criá-lo, vamos definir uma classe/entidade que será persistida. Essa unidade será compartilhada entre os projetos e estará acessível em todas as camadas. A **Listagem 3** exibe sua implementação.

Note que além dos campos Idade e Nome (linhas 11 e 12), o framework criará automaticamente o campo ID, que será a chave

primária da tabela. Observe que definimos também um método chamado CriarModelo (linhas 16 a 19). Ele está na mesma unidade da classe TSQLPessoa por motivos de conveniência para esse exemplo. Em uma aplicação real pode-se ter uma unidade específica para encapsular a criação do Modelo. Também é possível ter modelos diferentes para cada camada da aplicação, dessa forma, podemos ter entidades que são acessíveis somente em determinada camada.

A partir da classe do ORM, podemos criar um modelo e então inicializar o banco de dados SQLite. A **Listagem 4** mostra exatamente isto, por meio da implementação do método InicializarBancodeDados.

Listagem 3. Unit uPessoa, classe do ORM

```
01 unit uPessoa;

02 interface

03 uses
04   mORMot, SynCommons;

05 type
06  TSQLPessoa = class(TSQLRecord)
07  private
08    FIdade: Integer;
09    FNome: RawUTF8;
10  published
11    property Idade: Integer read FIdade write FIdade;
12    property Nome: RawUTF8 read FNome write FNome;
13  end;

14 function CriarModelo:TSQlModel;

15 implementation

16 function CriarModelo:TSQlModel;
17 begin
18   Result := TSQLModel.Create([TSQLPessoa]);
19 end;
```

Listagem 4. Inicialização do banco de dados

```
01 procedure TFrmServidor.InicializarBancoDeDados;
02 begin
03   // Cria o modelo da base de dados contendo a classe/tabela TSQLPessoa
04   ModeloDados := CriarModelo;

05  BaseDados := TSQLRestServerDB.Create(ModeloDados,
     'BasededadosSQLite.db3');
06  // Atualizará o esquema da base de dados de acordo com o modelo.
07   BaseDados.CreateMissingTables(0);
08 end;
```

A variável ModeloDados recebe a instância da criação do modelo através do método CriarModelo (linha 04). A seguir, a variável BaseDados recebe a instância do banco de dados através do método Create da classe TSQLRestServerDB (linha 05). Por último, na linha 07 é ilustrado o método CreateMissingTables que é responsável por criar novas tabelas ou campos na base de dados. Dessa forma, sempre que uma tabela ou campo for adicionado ao software, o framework irá acertar automaticamente o esquema da base de dados. Até o momento, o mORMot não trata campos renomeados ou deletados.

A partir da execução do método InicializarBancoDeDados (**Listagem 4**), podemos realizar operações na base de dados na instância do servidor, através do objeto BaseDados. Isso seria suficiente para uma aplicação Standalone, mas como o nosso objetivo é expor a camada de ORM através de HTTP/REST, temos ainda que criar a camada de comunicação HTTP.

Inicialização do servidor HTTP

O servidor HTTP será o responsável pela camada de comunicação REST/HTTP. A **Listagem 5** demonstra a implementação do método InicializarServidorHTTP.

Listagem 5. Inicialização do servidor HTTP

```
01 procedure TFrmServidor.InicializarServidorHTTP;
02 begin
03   ServidorHTTP := TSQLHttpServer.Create('8080', [BaseDados]);
04 end;
```

Inicializamos o servidor através da classe TSQLHTTPServer, passando como parâmetros a porta (8080) e a base de dados que queremos expor através de REST (objeto BaseDados).

As chamadas aos métodos ConfigurarLog, InicializarBancodeDados e InicializarServidorHTTP devem ser inseridas no evento OnCreate do formulário pois, desse modo, ao inicializar o aplicativo, o servidor já estará rodando. Logo na primeira vez ele criará automaticamente o arquivo da base de dados SQLite3.

A partir deste momento já é possível acessar o servidor via HTTP e o método GET, usando um browser através do endereço "http://localhost:8080/root/pessoa". Como ainda não temos informação na base de dados, o retorno deve ser um JSON contendo algumas informações da própria tabela, algo semelhante a: {"fieldCount":1,"values":["ID"],"rowCount":0}.

Aplicativo cliente

O aplicativo cliente irá demonstrar as seguintes funcionalidades:

- Conexão com o servidor;
- Requisição de um registro;
- Salvar/Atualizar/Deletar registros;
- Iterar sob os registros de uma tabela;
- Exibir todos os registros em um grid.

Dando início à aplicação cliente, criamos uma VCL Application através do menu File>New>VCL Forms Application – Delphi, utilizando para isso alguns componentes que vão nos possibilitar a demonstração das funcionalidades acima citadas. **A Figura 3** ilustra a interface da aplicação. A **Listagem 6** permite identificar os nomes e tipos de cada componente da tela, assim como três dos componentes do mORMot que usaremos (escopo private, linhas 25 a 27).

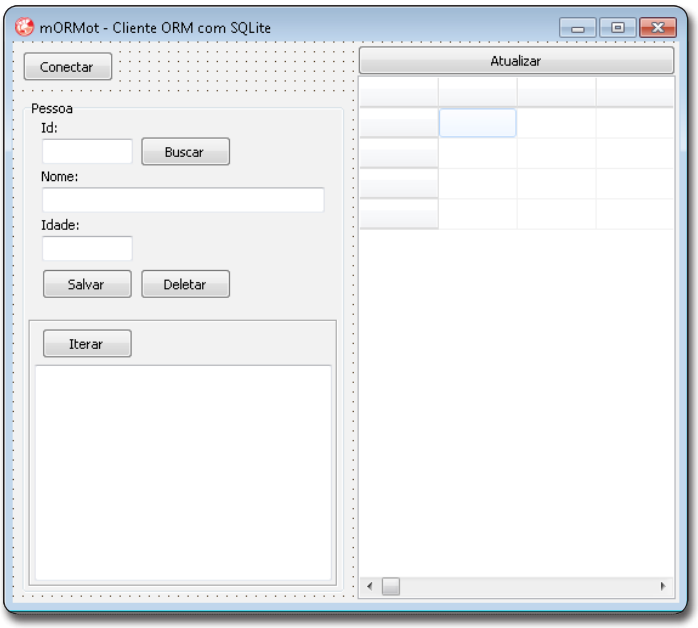

Figura 3. Aplicativo cliente

Listagem 6. Formulario cliente

```
01 uses
02  mORMotHttpClient, uPessoa, mORMotUI;

03  TFrmCliente = class(TForm)
04    BtnConectar: TButton;
05    GrpPessoa: TGroupBox;
06    EdtNome: TLabeledEdit;
07    EdtIdade: TLabeledEdit;
08    BtnSalvar: TButton;
09    EdtID: TLabeledEdit;
10    BtnDeletar: TButton;
11    BtnBuscar: TButton;
12    PnlGrid: TPanel;
13    GridPessoas: TDrawGrid;
14    BtnAtualizar: TButton;
15    Panel1: TPanel;
16    MemPessoas: TMemo;
17    BtnIterar: TButton;
18    procedure BtnConectarClick(Sender: TObject);
19    procedure BtnBuscarClick(Sender: TObject);
20    procedure BtnSalvarClick(Sender: TObject);
21    procedure BtnDeletarClick(Sender: TObject);
22    procedure BtnAtualizarClick(Sender: TObject);
23    procedure BtnIterarClick(Sender: TObject);
24  private
25    BaseDados: TSQLHttpClient;
26    Pessoas: TSQLPessoa;
27    Grid:TSQLTableToGrid;
28  end;
```

O componente BaseDados (linha 25) representa o ORM para a camada cliente, enquanto o objeto Pessoas (linha 26) será utilizado para guardar a lista de pessoas exibidas no grid. O objeto Grid (linha 27) é o responsável pelo controle do grid. O uso da classe TSQLTableToGrid facilita a exibição de dados do ORM, como veremos mais a diante.

A primeira implementação é do botão de conexão com o servidor. Nesse botão iremos realizar a conexão com o servidor e a busca das pessoas que serão exibidas no grid. A **Listagem 7** exibe esta tarefa.

Listagem 7. Botão conectar

```
01 procedure TForm4.BtnConectarClick(Sender: TObject);
02  procedure BuscarPessoas;
03  begin
04    Pessoas := TSQLPessoa.CreateAndFillPrepare(BaseDados, 'ID > 0');
05    Grid := TSQLTableToGrid.Create(GridPessoas, Pessoas.FillTable, BaseDados);
06  end;
07 begin
08  BaseDados := TSQLHttpClient.Create('localhost', '8080', CriarModelo);
09  if BaseDados.ServerTimeStampSynchronize then
10  begin
11    PnlGrid.Enabled := True;
12    GrpPessoa.Enabled := True;
13    BuscarPessoas;
14  end
15  else
16    ShowLastClientError(BaseDados,'Servidor não encontrado');
17 end;
```

Primeiramente instanciamos um objeto da classe TSQLHttpClient passando como parâmetros o endereço (localhost), a porta (8080) do servidor e o modelo de dados utilizado (linha 08), que em nosso caso, é o mesmo utilizado no servidor. Logo em seguida executamos o método ServerTimeStampSynchronize para verificar se o servidor está respondendo (linha 09). Como toda a comunicação é feita através de HTTP, não existe uma conexão constantemente ativa, como ocorre com os sockets TCP. Sempre que necessário, o framework irá estabelecer uma comunicação HTTP e realizar uma requisição REST ao servidor. Se o mesmo não estiver disponível naquele momento, será disparado um erro (linha 16). Caso o servidor esteja respondendo, vamos buscar a lista de pessoas para exibir no grid (linha 13). O método BuscarPessoas é interno e exclusivo da procedure principal. Nele é demonstrada a utilização do CreateAndFillPrepare (linha 04). Com ele é possível executar pesquisas personalizadas de determinada entidade. Além disso, também é permitido escolher entre trazer as informações de todos os campos (propriedades), ou somente alguns, o que permite uma resposta mais rápida já que o volume de dados será menor. CreateAndFillPrepare retornará uma lista de pessoas. Não precisamos de um tipo especial para trabalhar com essa lista, pois, o mesmo TSQLPessoa que é usado para um registro pode conter uma lista. Isso ficará mais claro depois de analisar a implementação da iteração sob os itens. Também é instanciado um objeto do tipo TSQLTableToGrid que vai formatar e alimentar o grid (linha 05). O grid terá algumas funcionalidades adicionadas pelo mORMot, como um novo visual, suporte a Unicode, colunas tipadas (campos booleanos serão exibidos como checkboxes, datas como texto), gerenciamento automático do tamanho das colunas, ordenação por coluna e etc.

A próxima implementação a ser feita é a do botão salvar, exibida na **Listagem 8**. Nela veremos como salvar e atualizar um registro através dos métodos Add e Update da classe TSQLHTTPClient.

Temos dois comportamentos (Salvar e Atualizar) no mesmo botão, há algumas validações de campos visando identificar o que deve ser realizado. Primeiramente, é checado se há um ID válido no campo EdtID (linha 23), caso haja, iremos usar uma das formas de buscar as informações de um determinado registro, que é através do construtor da classe TSQLPessoa. Ao passar como parâmetro o objeto BaseDados e o código do ID, estamos informando o framework que ele deve buscar as informações desse registro (linha 24). Caso encontre, o objeto pessoa irá conter todas as informações do registro em questão, caso contrário, será criado um objeto vazio. Também é possível requisitar somente alguns campos/propriedades com o intuito de economizar banda de dados.

Listagem 8. Botão Salvar

```
01 procedure TForm4.BtnSalvarClick(Sender: TObject);
02 var
03   Pessoa: TSQLPessoa;
04   procedure NovoRegistro;
05   begin
06     if BaseDados.Add(Pessoa, true) > 0 then
07     begin
08       ShowMessage('Salvo');
09       if StrToIntDef(EdtID.Text,0) = 0 then
10         EdtID.Text := IntToStr(Pessoa.ID);
11     end
12     else
13       ShowMessage('O registro não foi salvo.
               Erro:'+baseDados.LastErrorMessage);
14   end;
15   procedure AtualizarRegistro;
16   begin
17     if BaseDados.Update(Pessoa) then
18       ShowMessage('Registro atualizado')
19     else
20       ShowMessage('O registro não foi salvo.
               Erro:'+baseDados.LastErrorMessage);
21   end;
22 begin
23   if StrToIntDef(EdtID.Text,0) > 0 then
24     Pessoa := TSQLPessoa.Create(BaseDados, StrToInt(EdtID.Text))
25   else
26     Pessoa := TSQLPessoa.Create;
27   try
28     Pessoa.Nome := EdtNome.Text;
29     Pessoa.Idade :=  StrToInt(EdtIdade.Text);
30     if Pessoa.ID > 0 then
31       AtualizarRegistro
32     else
33       NovoRegistro;
34   finally
35     Pessoa.Free;
36   end;
37   Grid.Refresh;
38 end;
```

Se o usuário não informou nenhum ID, é um registro novo, deste modo é criado o objeto pessoa sem passar nenhum parâmetro no construtor, o que criará um objeto TSQLPessoa vazio (linha 26).

Após alimentar as propriedades com os valores informados na tela, precisamos inserir ou atualizar o registro. Se o ID for zero (linha 33), será chamada a procedure interna "NovoRegistro" especificada entre as linhas 04 a 14. Observe que a inserção de uma nova pessoa será realizada por meio do método "BaseDados.Add" (linha 06). Caso o ID possua um valor superior a zero, será necessário realizar o processo de atualização do registro, que é realizado através do procedimento "AtualizarRegistro" descrito na linha 31. Assim como o método NovoRegistro, esta procedure é de uso exclusivo neste processo, não estando visível aos demais métodos de nossa aplicação. Sua implementação encontra-se entre as linhas 15 a 21. Vale lembrar que o responsável por esta atualização é especificamente o comando "BaseDados.Update" (linha 17).

O comando Add retorna o ID do novo registro em caso de sucesso, enquanto o comando Update retorna um booleano com valor True caso o registro tenha sido atualizado e False caso contrário. Em casos de falha, podemos verificar as propriedades LastErrorMessage, LastErrorException e LastErrorCode (linhas 13 e 20) para identificar a causa do problema.

Após inserir alguns registros será possível acessar o servidor pelo browser através da URL "http://localhost:8080/root/pessoa/1", onde o valor 1 corresponde ao ID da pessoa que está sendo requisitada. O retorno será um JSON contendo o conteúdo da classe TPessoa, algo como:

```
{"ID":1,"Idade":26,"Nome":"Roberto"}.
```

A implementação do evento OnClick do botão de busca é mostrada na **Listagem 9**. Nela são detalhadas outras formas de se requisitar informações de um determinado registro. Neste caso, utilizamos o método Retrieve (linha 07) da classe TSQLHTTPClient para requisitar uma pessoa com o ID informado no campo EdtId. Como parâmetros passamos um objeto pessoa em branco e o ID que estamos buscando. Caso o registro com esse ID não exista na base de dados, o método Retrieve retornará false e o objeto permanecerá em branco, caso contrário os valores são automaticamente passados aos componentes visuais correspondentes.

O próximo evento é o OnClick do botão Deletar que encontra-se ilustrado por meio da **Listagem 10**. Observe que para deletar um registro não precisamos instanciar nenhum objeto, bastando simplesmente passar o ID correspondente para o método Delete (linha 05), assim como, a classe relativa à entidade que se quer deletar, nesse caso, a própria classe TSQLPessoa. O retorno é booleano e segue a mesma regra do Update. Nas linhas 08 a 10 os componentes visuais simplesmente são limpos e a Grid atualizada, conforma a linha 15.

Para iterar (percorrer) entre vários registros, usamos a lista de pessoas que estamos exibindo no Grid (**Listagem 11**). Através do método FillOne (equivalente ao Next de um Dataset – linha 03) é possível navegar entre todos os itens da lista.

Para atualizar uma lista carregada em um grid, basta chamar o método Refresh da classe TSQLTableToGrid. Uma nova requisição REST será enviada ao servidor para verificar a necessidade de se atualizar os dados. Caso haja alguma mudança em relação aos dados, eles serão retransmitidos para o cliente. Sendo assim, basta utilizar o método conforme sugerido na **Listagem 12**.

VOCÊ TAMBÉM
ESTÁ BOIANDO
NESTA SOPA?
CSS
HTML
JPA
TDD
JSON
MYSQL

Listagem 9. Botão Buscar

```
01 procedure TForm4.BtnBuscarClick(Sender: TObject);
02 var
03  Pessoa: TSQLPessoa;
04 begin
05  Pessoa := TSQLPessoa.Create;
06  try
07   if not BaseDados.Retrieve(StrToInt(EdtId.Text), Pessoa) then
08    ShowMessage('Pessoa não encontrada')
09   else
10   begin
11    EdtNome.Text := Pessoa.Nome;
12    EdtIdade.Text := IntToStr(Pessoa.Idade);
13   end;
14  finally
15   Pessoa.Free;
16  end;
17 end;
```

Listagem 10. Botão Deletar

```
01 procedure TForm4.BtnDeletarClick(Sender: TObject);
02 begin
03  if StrToIntDef(EdtId.Text,0) > 0  then
04  begin
05   if BaseDados.Delete(TSQLPessoa, StrToInt(EdtId.text)) then
06   begin
07    ShowMessage('Registro '+EdtId.text+' deletado');
08    EdtId.Clear;
09    EdtNome.Clear;
10    EdtIdade.Clear;
11   end;
12  end
13  else
14   ShowMessage('Informe o ID');
15  Grid.Refresh;
16 end;
```

Listagem 11. Botão Iterar

```
01 procedure TForm4.BtnIterarClick(Sender: TObject);
02 begin
03  while Pessoas.FillOne do
04  begin
05   MemPessoas.Lines.Add('ID: '+IntToStr(Pessoas.Id) +
     '; Nome: '+Pessoas.Nome + '; Idade: '+IntToStr(Pessoas.Idade));
06  end;
07 end;
```

Listagem 12. Botão Atualizar

```
01 procedure TForm4.BtnAtualizarClick(Sender: TObject);
02 begin
03  Grid.Refresh;
04 end;
```

SOA baseado em interfaces

Um dos recursos mais fantásticos do framework é a sua engine SOA que trabalha baseada em interfaces. No exemplo implementaremos um serviço e publicaremos em uma interface REST, e a seguir, consumiremos através da aplicação cliente anteriormente criada.

A interface que queremos expor está listada no código da **Listagem 13**. Assim como a entidade do ORM, essa interface é compartilhada entre os projetos e camadas. Note que é compartilhada somente a **interface** e não a implementação.

A implementação dessa interface é conhecida somente pelo servidor. A aplicação cliente não irá conhecê-la, o que nos permitirá desacoplar totalmente as aplicações que consomem os serviços de suas implementações. Com isso, é possível alterar o código dos serviços sem a necessidade de recompilar e redistribuir as aplicações clientes. Para exemplificar, faremos um caso com uma classe TCalculadora que será responsável por implementar a interface definida anteriormente (**Listagem 14**).

Listagem 13. Interface calculadora

```
01 unit uCalculadora;

02 type
03  ICalculadora = interface(IInvokable)
04   ['{211B45A1-B39B-4E8A-A12E-F3F6A88A54DC}']
05   function Somar(n1,n2: integer): integer;
06 end;
```

Listagem 14. Implementação da calculadora

```
01 unit uCalculadoraImpl;

02 interface

03 uses uCalculadora;

04 type
05  TCalculadora = class(TInterfacedObject, ICalculadora)
06   function Somar(n1,n2: integer): integer;
07  end;

08 implementation

09 function TCalculadora.Somar(n1, n2: integer): integer;
10 begin
11  Exit(n1+n2);
12 end;

13 end.
```

Após realizar a implementação da classe TCalculadora, conforme a **Listagem 14**, é necessário fazer algumas alterações no servidor. Para isso necessitaremos de dois novos objetos, ModeloServicos do tipo TSQLModel e ServidorServicos do tipo TSQLRestServer, que podem ser declarados na sessão private do formulário, assim como a variável BancoDados por exemplo, utilizada anteriormente.

A **Listagem 15** mostra o código responsável por realizar a inicialização do servidor de serviços, chamado InicializarServiços.

No código da **Listagem 15** criamos um modelo vazio (linha 03), somente para especificar outro caminho para as requisições do tipo serviço. Ao invés de utilizar o endereço "localhost:8080/root", como ocorre para o ORM, apontaremos para "localhost:8080/services".

A última linha (05) é responsável por registrar o serviço, onde informamos que a classe TCalculadora será utilizada como implementação da interface ICalculadora, muito semelhante ao que é feito nos frameworks de injeção de dependência. O parâmetro sicShared define a forma como esse serviço será gerenciado pelo servidor.

ASSIM, O MERCADO TE ENGOLE!

PARAR DE SE ATUALIZAR PODE SER FATAL!

O mercado muda a todo momento e você precisa estar atento sempre. Seja um assinante MVP e devore tudo o que há de mais relevante no mercado de TI.

Invista na sua carreira!

TENHA ACESSO A:

- +DE 260 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 7.850 VÍDEO-AULAS

POR APENAS 59,90 MENSAIS

MVP
MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

DevMedia | Acesse: www.devmedia.com.br/mvp

Neste caso haverá uma instância única para todos os clientes. É possível escolher entre opções que permitem gerenciar uma instância por Thread, por Cliente, Sessão entre outras.

O próximo passo é alterar o método InicializarServidorHttp já que agora temos além do ORM, um servidor de serviços para expor via REST. Observe o método na **Listagem 16**. Nela adicionamos o servidor de serviços (Objeto ServidorServicos) a lista de servidores que serão expostos via REST.

O evento OnCreate do formulário também deverá sofrer mudanças para executar o método de inicialização do servidor de serviços, conforme a **Listagem 17**. Após estes passos estamos prontos para fazer o primeiro teste e implementar a aplicação cliente.

A implementação em específico não tem como objetivo facilitar o uso dos serviços via AJAX, mas ainda assim é possível. Através do método GET podemos utilizar a seguinte URL no browser: "http://localhost:8080/services/calculadora.somar?+%5B+1%2C2+%5D" para usar o método somar do servidor Calculadora. A expressão "+%5B+1%2C2+%5D" será decodificada como [1,2], logo, o resultado esperado é "{"result":[3],"id":0}". Para o método POST o uso é mais simples, sendo possível passar parâmetros através de JSON. Há maneiras de facilitar a utilização via GET, com a utilização de parâmetros no padrão "/calculadora.somar?a=1&b=2". Como o nosso objetivo é usar os serviços em um cliente Delphi, usamos a implementação mais simples.

Listagem 15. Método Inicializar Serviços

```
01 procedure TFrmServidor.InicializarServicos;
02 begin
03   ModeloServicos := TSQLModel.Create([],'services');
04   ServidorServicos := TSQLRestServerFullMemory.Create(ModeloServicos);
05   ServidorServicos.ServiceRegister(TCalculadora,[TypeInfo(ICalculadora)],
     sicShared);
06 end;
```

Listagem 16. Método Inicializar Servidor HTTP

```
01 procedure TFrmServidor.InicializarServidorHTTP;
02 begin
03   ServidorHTTP := TSQLHttpServer.Create('8080', [BaseDados,
     ServidorServicos]);
04   Label1.Caption := 'Servidor rodando na porta 8080'
05 end;
```

Listagem 17. Evento OnCreate do formulário

```
01 procedure TFrmServidor.FormCreate(Sender: TObject);
02 begin
03   ConfigurarLog;
04   InicializarBancoDeDados;
05   InicializarServicos;
06   InicializarServidorHTTP;
07 end;
```

Para a aplicação cliente também necessitaremos de dois novos objetos declarados na sessão private do formulário. O primeiro é o objeto "ModeloServicos" do tipo TSQlModel e o segundo é chamado de "Servicos", do tipo TSQLHttpClient. Para inicializar o servidor, pode-se criar um método conforme o sugerido no evento OnClick do botão ConectarServcios, na **Listagem 18**.

Feito isso já é possível acessar o serviço que criamos através do container de serviços disponibilizado pelo mORMot. A **Listagem 19** mostra a implementação de um método consumindo este serviço.

Listagem 18. Evento OnClick

```
01 procedure TFrmCliente.BtnConectarServicosClick(Sender: TObject);
02 begin
03   ModeloServicos := TSQLModel.Create([], 'Services');
04   Servicos := TSQLHttpClient.Create('Localhost', '8080', ModeloServicos);
05   Servicos.ServiceRegister([TypeInfo(ICalculadora)],sicShared);
06 end;
```

Listagem 19. Evento OnClick do botão Somar

```
01 procedure TFrmCliente.BtnSomarClick(Sender: TObject);
02 var
03   Calculadora: ICalculadora;
04   Resultado: Integer;
05 begin
06   Servicos.Services['Calculadora'].Get(Calculadora);
07   Resultado := Calculadora.Somar(StrToInt(EdtValor1.Text),
           StrToInt(EdtValor2.text));
08   ShowMessage(IntToStr(Resultado));
09 end;
```

Para acessar o serviço criamos uma variável do tipo ICalculadora e invocamos o método Services.Get (linha 06) para obter uma instância do objeto que implementa essa interface. Há ainda outras maneiras de invocar o serviço. É importante novamente salientar que a aplicação cliente não conhece a implementação da interface que fizemos na unit uCalculadoraImpl. Ou seja, a classe TCalculadora se quer é compilada com a aplicação cliente. O framework irá gerar um objeto "fake" contendo os métodos da interface em questão, e, quando uma chamada for realizada (ex: Somar), será realizada uma requisição REST ao servidor, onde o processamento será feito.

Após obter o objeto "fake" que implementa a interface ICalculadora, podemos utilizá-lo normalmente, como usaríamos qualquer outro objeto. Toda comunicação REST é encapsulada pelo framework.

Esse recurso permite a utilização de serviços REST de uma forma extremamente simples, e é um dos maiores diferenciais do mORMot, sendo inclusive mais simples que as opções disponíveis para JAVA e .NET.

Conclusão

É importante elaborar uma arquitetura consistente para que o software consiga crescer e evoluir conforme as necessidades. Existem muitos problemas para serem resolvidos quando se está tentando melhorar a arquitetura de um software.

O mORMot fornece ferramentas e funcionalidades que facilitam muito algumas decisões e implementações, permitindo que você foque nos problemas do seu domínio. Não há porque gastar

tempo e dinheiro implementando soluções que já estão prontas e livres para uso. Sendo assim, é possível economizar muito tempo utilizando os recursos do framework, assim como também é possível melhorar e muito arquiteturas antigas aos poucos, sem a necessidade de grandes e problemáticas migrações.

Seria impossível demonstrar todos seus recursos em um único artigo. Suas mais de 1000 páginas de documentação dão uma ideia da dimensão do projeto, e embora simples, esse exemplo demonstra o quão fácil é sua utilização.

Vários problemas comuns de aplicações arquitetadas com o DataSnap são resolvidos de uma forma muito simples no mORMot, como as bibliotecas JSON, autenticação, log, testes unitários e etc.

Ainda há alguns pontos a evoluir, como por exemplo, um melhor suporte a mapeamento de objetos para tabelas antigas de uma base de dados existente, no entanto, o trabalho que está sendo realizado nesse projeto é frequente.

O mORMot traz consigo uma tecnologia que apesar de ser comum em plataformas como Java e .NET, ainda não estava disponível para o Delphi. Muitas das suas funcionalidades foram baseadas em comportamentos destas linguagens citadas, e o objetivo nunca foi copiá-las, e sim fazer um framework com as características da linguagem Object Pascal, com as vantagens do Delphi e com as mesmas funcionalidades, ou até melhores que em outras plataformas.

Em um ambiente dominado pelo RAD, enfim temos uma solução orientada a objetos de alto nível e com um grande potencial.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Autor

Roberto Schneiders

roberto.schneiders@gmail.com - robertocschneiders.wordpress.com/

Bacharel em sistemas de informação pela UNOESC-SMO. Atua como analista/programador de sistemas pela Sysmo Sistemas (www.sysmo.com.br). Co-fundador da Eletrone (facebook.com/Eletrone-Brasil). Trabalha com Delphi e Java.

Links:

Documentação do mORMot
http://synopse.info/files/pdf/Synopse%20mORMot%20Framework%20SAD%201.18.pdf

Blog do Roberto Schneiders
http://robertocschneiders.wordpress.com/

Blog da Synopse
http://blog.synopse.info

Página do mORMot
http://synopse.info/fossil/wiki?name=SQLite3+Framework

Fórum do mORMot
http://synopse.info/forum/viewforum.php?id=2

Roadmap do mORMot
http://synopse.info/fossil/wiki?name=RoadMap

Timeline do mORMot
http://synopse.info/fossil/timeline

Don't be confused by our little mORMot !
http://blog.synopse.info/post/2012/09/10/Don-t-be-confused-by-our-little-mORMot-!

Two videos about EXTjs client of mORMot server
http://blog.synopse.info/post/2013/04/02/Two-videos-about-EXTjs-client-of-mORMot-server

Página de ajuda de instalação do mORMot
http://synopse.info/fossil/wiki?name=Get+the+source

Boas práticas para redução de acoplamento

Utilização de interfaces, injeção de dependências e container de inversão de controle

Quando desenvolvemos um sistema, seja ele orientado a objetos ou não, o que mais queremos é que o mesmo seja fácil de manter. Esse é o pensamento inicial de cada projeto. Porém, com o passar do tempo, redução de prazos e custos, deixamos muitas vezes de pensar na arquitetura que irá permitir essa redução de trabalho de manutenção no futuro e nos preocupamos apenas no "entregar software". O não pensar na arquitetura geral produz rotinas altamente acopladas, ou seja, rotinas que dependem, conhecem, tanto uma a outra que qualquer alteração em uma, implica em uma cascata de ajustes. Isso é acoplamento.

Existe uma boa prática comum no desenvolvimento orientado a objetos que permite reduzir essa forte dependência, mesmo que já exista um sistema legado é possível através de refactoring (**BOX 1**) ir melhorando partes críticas. Essa boa prática consiste em encapsular o que varia através de interfaces. Para clarificar esse processo vamos abordar um exemplo fictício, que mostrará partes de um pseudocódigo empregado para a solução do problema apresentado.

Utilizando interfaces

Pensemos em um exemplo bem simples, temos um site onde os clientes realizam pedidos e ao final é emitida uma cobrança. Vamos imaginar que em um primeiro momento os pagamentos serão realizados exclusivamente através de Boletos Bancários. Começamos então criando uma classe que será responsável por esse tipo de pagamento, veja na **Listagem 1**.

Os pedidos realizados também são representados em classes, no caso, a classe TPedido (**Listagem 2**). Temos um método chamado FecharPedido, responsável pelas ações finais que um pedido sofre ao ser finalizado, que neste modesto exemplo, irá simplesmente emitir um Boleto. Aqui temos a primeira diferença da abordagem orientada a objetos em relação à procedural.

Resumo DevMan

Porque esse artigo é útil:

Ao desenvolver um software, não temos de início todas as funcionalidades que o mesmo deverá executar. Por mais que a fase de análise seja detalhada para levantar todos os requisitos, uma única coisa é certa. Esses irão mudar. Construir um aplicativo pensando em todas essas possíveis mudanças fica fácil quando se pensa na arquitetura a utilizar. Nesse artigo três boas práticas são abordadas, o uso de interfaces para isolamento das regras de negócio sujeitas a mudanças, injeção de dependência que reduz o quanto uma classe irá conhecer de outra e a aplicação de um container de inversão de controle, que flexibiliza o uso das interfaces como serviços de um software. Aplicando inicialmente essas três recomendações, qualquer sistema legado dá um passo para se tornar melhor, mais fácil de ser modificado.

BOX 1. Refactoring

Um dos grandes problemas encontrados em sistemas de software é evitar a deterioração tão comum dos programas durante o ciclo de vida do sistema. Essa deterioração é, geralmente, causada por modificações com objetivos de curto prazo ou por alterações realizadas sem a clara compreensão da concepção do sistema. A refatoração (do inglês Refactoring) é o processo de modificação de software para melhorar o entendimento do código sem alterar suas funcionalidades, removendo duplicidades, melhorando a comunicação da equipe, simplificando e flexibilizando rotinas, facilitando a manutenção e evitando a inclusão de novos defeitos, aumentando assim a qualidade do produto.

O livro mais importante sobre o assunto é Refatoração - Aperfeiçoando o Projeto de Código Existente de Martin Fowler, que apresenta um catálogo com 72 refatorações, explicando seus conceitos e motivações, além de descrever detalhadamente cada uma das refatorações apresentadas.

O autor argumenta que, com treinamento apropriado, um profissional de computação pode transformar um projeto de baixa qualidade num código robusto e bem projetado, sem modificar seu comportamento observável. Outro bom argumento para a utilização dessa técnica é que qualquer um pode escrever código que um computador consiga entender, enquanto bons programadores escrevem código que pessoas conseguem entender.

A refatoração modifica um programa em pequenos passos. Assim, se um erro é cometido, torna-se mais fácil consertá-lo. Algumas oportunidades são apropriadas para aplicar refatoração. Deve-se refatorar quando for necessário construir código parecido com outro já existente, quando acrescentar novas funções, quando necessita consertar uma falha ou enquanto se revisa um código.

Enquanto numa abordagem tradicional teríamos somente uma tabela representando as informações de um pedido, temos na orientação a objetos uma classe, constituída de dados e comportamento. É por isso que muitos arquitetos afirmar que através de classes é possível representar de uma forma melhor os problemas de negócio que um sistema tenta resolver. Em nosso exemplo o comportamento de um pedido está exposto no método FecharPedido.

Basta chamar o método FecharPedido da classe TPedido para que o boleto de cobrança seja emitido, veja como fica a chamada na **Listagem 3**. Não precisamos detalhar aqui o resultado que esse código irá exibir.

Listagem 1. Classe TBoleto

```
TBoleto = class
public
  procedure Cobrar(Valor: Currency);
end;
procedure TBoleto.Cobrar(Valor: Currency);
begin
  Writeln('Emitir boleto no valor de ' +
    FloatToStrF(Valor,ffCurrency,15,2));
end;
```

Listagem 2. Classe TPedido

```
TPedido = class
private
  FBoleto: TBoleto;
public
  constructor Create;
  procedure FecharPedido(Valor: Currency);
end;
constructor TPedido.Create;
begin
  inherited;
  FBoleto := TBoleto.Create;
end;

procedure TPedido.FecharPedido(Valor: Currency);
begin
  FBoleto.Cobrar(Valor);
end;
```

Listagem 3. Emitindo o boleto de cobrança

```
var
  pedido: TPedido;
begin
  try
    pedido := TPedido.Create;
    pedido.FecharPedido(200);
    ReadLn;
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Isso funciona muito bem enquanto nosso site estiver trabalhando com boleto bancário. Mas e se a regra de negócio é alterada e for necessário implementar outras formas de pagamento? Como se pode notar na **Listagem 2**, o boleto é criado dentro da classe TPedido, isso significa que toda vez que quisermos implementar uma nova forma de pagamento, teremos que alterar essa classe. Quando temos este tipo de dependência, dizemos que a classe TPedido está fortemente acoplada à classe TBoleto. Quando temos classes fortemente acopladas, elas não podem ser substituídas sem que suas implementações sejam alteradas. Aqui o alto acoplamento está bem representado. Em um exemplo simples podemos notar um problema bem grave, imaginemos então em um sistema real e mais complexo como suas tarefas/funcionalidades podem se emaranhar.

Para remediar a situação utilizamos interfaces para diminuir esse acoplamento entre as classes, encontrando o que varia e encapsulando. No projeto de exemplo o que está variando? A forma de pagamento. Dessa forma é criada uma interface para representar as formas de pagamento que o site irá oferecer. Veja na **Listagem 4**.

Listagem 4. Interface IFormaPagamento

```
IFormaPagamento = interface
  ['{083BAF04-A68D-4148-8F7B-A10E44667CC3}']
  procedure Cobrar(Valor: Currency);
end;
```

Cada forma de pagamento deve implementar a interface IFormaPagamento. Na **Listagem 5** vemos a classe TBoleto fazendo essa implementação. Ela é descendente de TInterfacedObject, uma classes especial do Delphi para implementação de interfaces, assim, objetos instanciados de classes herdadas a partir de TInterfacedObject não necessitam ser destruídos, pois possuem um contador de referências interno que automaticamente os destrói quando esse contador chega a zero. Além disso, esta classe TInterfacedObject tem um mecanismo para trabalhar com multithread que previne que um objeto seja destruído quando ainda não terminou de ser criado.

Então, teoricamente usando interfaces e criando nossas classes como descendentes de TInterfacedObject não precisamos nos preocupar com a destruição dos objetos e nem com Memory Leaks, que é o termo dado para um objeto que foi criado em tempo de execução, contudo, após seu devido uso e descarte, não é destruído – eliminado da memória.

A classe TPedido agora não precisa mais ter uma dependência direta à classe TBoleto, mas sim uma referência à interface IFormaPagamento, que atua como um objeto mais genérico de cobrança, como mostra a **Listagem 6**.

Note que além de termos um objeto genérico que irá representar a forma de pagamento, não estamos mais instanciando o Boleto no construtor do Pedido. O Boleto, ou qualquer outra forma de pagamento, será passado ao construtor. Dessa forma, a "dependência" entre as classes TPedido e TBoleto é "injetada" no construtor da classe TPedido.

Injeção de dependência

Aqui entra outro conceito que garante o baixo acoplamento, a chamada injeção de dependência. Ao remover da classe Pedido a responsabilidade da criação das formas de pagamento, ela está

flexível o suficiente para aceitar qualquer nova forma de pagamento, desde que a classe representada implemente a interface IFormaPagamento.

Assim, conseguimos criar outras formas de pagamento sem termos que modificar coisa alguma na classe TPedido. Por exemplo, a **Listagem 7** mostra como ficaria uma nova forma de pagamento, o cartão de crédito representado pela classe TCartaoCredito.

Listagem 5. TBoleto implementando IFormaPagamento

```
TBoleto = class(TInterfacedObject, IFormaPagamento)
public
  procedure Cobrar(Valor: Currency);
end;

procedure TBoleto.Cobrar(Valor: Currency);
begin
  Writeln('Emitido boleto no valor de ' +
     FloatToStrF(Valor,ffCurrency,15,2));
end;
```

Listagem 6. TPedido usando o IFormaPagamento

```
TPedido = class
private
  FFormaPagamento: IFormaPagamento;
public
  constructor Create(formaPagamento: IFormaPagamento);reintroduce;
  procedure FecharPedido(Valor: Currency);
end;

constructor TPedido.Create(formaPagamento: IFormaPagamento);
begin
  inherited Create;
  FFormaPagamento := formaPagamento;
end;

procedure TPedido.FecharPedido(Valor: Currency);
begin
  FFormaPagamento.Cobrar(Valor);
end;
```

Listagem 7. Classe TCartaoCredito

```
TCartaoDeCredito = class(TInterfacedObject, IFormaPagamento)
public
  procedure Cobrar(Valor: Currency);
end;

procedure TCartaoDeCredito.Cobrar(Valor: Currency);
begin
  Writeln('Transacao no valor de: ' + FloatToStrF(Valor,ffCurrency,15,2) +
       ' autorizada pela operadora.');
end;
```

Já a **Listagem 8** mostra como fica a criação dos Pedidos e a chamada do método FecharPedido.

Simplesmente é passada uma instância de cada forma de pagamento para cada pedido criado. Isso mostra que podemos criar qualquer outra forma sem que haja a necessidade de modificarmos a classe TPedido.

Listagem 8. Injetando a dependência

```
var
  pedido1: TPedido;
  pedido2: TPedido;
begin
try
    pedido1 := TPedido.Create(TBoleto.Create);
    pedido1.FecharPedido(200);
    pedido2 := TPedido.Create(TCartaoDeCredito.Create);
    pedido2.FecharPedido(150);
    ReadLn;
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
end;
end.
```

Inversão de controle

A inversão de controle (IoC) é um princípio de design que favorece o desacoplamento de funcionalidades. Seu uso permite a criação de softwares mais testáveis, flexíveis e que podem ser estendidos facilmente. Em um primeiro momento ele pode ser muito parecido com o princípio da injeção de dependência, pois sua ideia básica é que durante o desenvolvimento deveriam especificar quais **serviços** são necessários por um **componente**, ao invés de criar uma instância diretamente. A parte parecida com a injeção de dependência é a determinação de não criar objetos de forma direta. Mas sua diferença é bem sutil. São mencionados serviços e componentes.

Um serviço nada mais é do que aquela rotina de um projeto que pode sofrer alteração constantemente e por isso, é encapsulada em uma interface. Já o componente é a classe que faz uso do serviço. Se observarmos os exemplos das **Listagens 7** e **8** temos um uso conjunto de IoC e injeção de dependência e com isso conseguimos controlar o comportamento da classe TPedido através de quem a instancia. Esse procedimento também está relacionado a outro princípio de modelagem, o princípio de Hollywood, que diz, "não nos chame, nós chamamos você". Os benefícios desse conjunto de abordagem são:

- Testabilidade: Podemos criar mocks (**BOX 2**) durante um teste;
- Flexibilidade: Em tempo de execução é possível disponibilizar várias implementações de um mesmo serviço;
- Extensibilidade: É fácil criar implementações variadas para um serviço.

BOX 2. Mocks

Mock é o conceito de criar objetos que imitam objetos reais para efeito de teste. Por exemplo, temos no sistema uma rotina que envia e-mails de confirmação de compra, ou seja, assim que uma compra é efetuada seu respectivo e-mail é enviado ao comprador. Em termos técnicos poderíamos ter uma classe TCompra que utiliza uma interface IEmail. Ao testar especificamente a rotina de compra ela não pode depender do comportamento do envio de e-mail padrão. Dessa forma, é criada outra implementação de IEmail que apenas "simula" o envio, ou seja, essa simulação nunca dará errado e não comprometerá os testes de TCompra. Existem frameworks que facilitam esse tipo de teste. Para Delphi temos o Delphi-Mocks, veja seção **Links**.

Juntamente com o IoC surgiu outro conceito para facilitar seu uso. O container IoC, que basicamente nada mais é do que uma classe que centraliza a responsabilidade da resolução das interfaces. Isso quer dizer que, ao invés de no próprio código instanciar uma determinada implementação, uma solicitação é feita a esse container e o mesmo retornaria a implementação registrada para determinada interface. Existe para Delphi o framework Spring4D (seção **Links**), vamos abordar suas funcionalidades.

Spring4D

O Spring4D é um projeto open source, para versão Delphi 2010 e superiores, que consiste de diferentes módulos. Ele inclui um framework de injeção de dependência, representado na Unit Spring.IoC, simples que suporta:

- O registro/injeção de dependência via programação ou atributos;
- Registro no estile fluente;
- Interfaces fortemente tipadas que permitem a verificação em tempo de compilação;
- Tipos pré-definidos de tempo de vida para os objetos (Singleton/Transient/SingletonPerThread/Pooled);
- Injeção de dependência de construtores, métodos, propriedades e campos;
- Um serviço pode ser registrado por diferentes componentes, sendo identificado por nome;
- Um componente pode implementar uma série de serviços;
- Verificação de referência circular;
- Escolha de como ativar um componente, por reflexão ou delegação;

Também disponibiliza uma biblioteca de criptografia, existente na Unit Spring.Cryptography que suporta os seguintes algoritmos:

- Hash (MD5, SHA-1/256/384/512);
- Checksum (CRC-16, CRC-32);
- Algoritmos simétricos (DES, Triple-DES, CipherMode, PaddingMode e IV);
- Gerador randômico de números

Sua Unit Spring.Collections expõe interfaces de coleções genéricas conhecidas por desenvolvedores .Net, como ICollection<T>, IList<T>, IDictionary<TKey, TValue>. Essas interfaces são compatíveis com a interface System.IEnumerable<T>.

Compilando o Spring4D

Uma vez feito o download do código fonte do projeto e ter descompactado seu conteúdo podemos compilar o projeto. A versão disponível possui um aplicativo que realiza o trabalho de compilação para o RAD Studio 2010, XE e XE2. Esse aplicativo encontra-se na pasta Spring e chama-se Build.exe. Ao executá-lo basta escolher a versão do Delphi e o tipo de configuração (Debug/Release) e clicar no botão Build (**Figura 1**).

Caso o Delphi a ser utilizado é de uma versão mais recente, é necessária a compilação manual. Para isso carregamos no IDE o arquivo Spring4D.groupproj, localizado em \Spring\Packages\DelphiXE2. Esse arquivo groupproj agrupa os projetos que compõem o Spring4D (**Figura 2**).

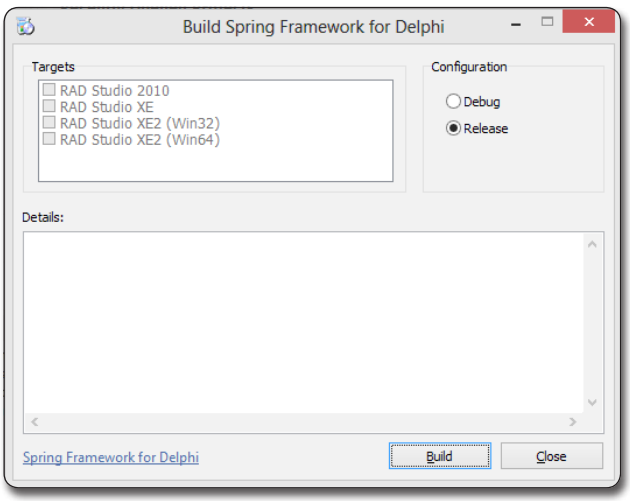

Figura 1. Ferramenta de compilação Build.exe

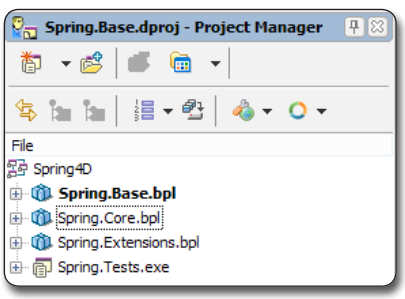

Figura 2. Projetos Spring4D

A versão disponibilizada possui uma incompatibilidade com versões do Delphi superiores a versão XE 2 no projeto Spring .Extensions. Isso acontece porque nela é utilizado um recurso que não é mais suportado, o recurso de chamar um construtor utilizando variáveis de instância. Alterar esse projeto é um esforço que não é preciso nesse momento, assim, realizamos o Build (clicar com o botão direito nos projetos e escolher Build) apenas de Spring.Base e Spring.Core. O resultado de ambas as formas de compilação estará disponível em Spring\Library\DelphiXE2\Win32\Debug.

O projeto de exemplo

Para mostrar as facilidades que um container IoC traz, um projeto é criado no Delphi XE3 (File>New>VCL Forms Application). Ao salvá-lo foram mantidos os nomes padrão. O aplicativo irá demonstrar como mudar a implementação de exibição de mensagens sem afetar quem as executa. Digamos que todas as mensagens de diálogo do projeto foram construídas utilizando a função nativa MessageBox, porém será necessário trocar o MessageBox por outra implementação particular.

Antes de já começar a sair programando, algumas decisões de arquitetura foram analisadas. Por exemplo, uma boa prática já

citada é encapsular o que varia. Nesse exemplo o que irá variar é a exibição das mensagens. Outro ponto, é que o conhecimento de exibição de mensagens não deve ser de conhecimento de todas as funcionalidades do sistema, mas sim, de uma única classe que atua como um serviço de mensagem. Com esse raciocínio em mente a exibição de mensagens foi abstraída para um serviço que é independente em relação à tecnologia. Uma nova Unit foi adicionada, chamada de IMensagem.pas. Nela foi criada a interface IServicoMensagem, como mostra a **Listagem 9**.

Listagem 9. Serviço de Mensagem

```
01 unit IMensagem;
02
03 interface
04
05 type
06  TBotoes = (btnOk, btnOkCancel, btnSimNao);
07  TTipoAlerta = (taInformacao, taAlerta, taErro);
08  TRetornoMensagem = (rmOk, rmCancel, rmSim, rmNao);
09
10  IServicoMensagem = Interface
11   ['{F86C661E-EFF5-44D3-968B-ECA17AD88032}']
12    function Exibir(Mensagem: string;
13              TituloMensagem: string;
14              TipoMensagem: TTipoAlerta;
15              Botoes: TBotoes): TRetornoMensagem;
16  End;
17
18 implementation
19
20 end.
```

Para deixar o serviço independente não apenas a exibição foi abstraída mas características de uma caixa de diálogo também. Por exemplo, a enumeração TBotoes representa quais botões serão exibidos, enquanto que TTipoAlerta indica qual a severidade da mensagem. Assim, será possível personalizar a exibição de acordo com sua importância. Por fim, o retorno da mensagem está representado na enumeração TRetornoMensagem.

A implementação básica

Uma nova Unit foi adicionada ao projeto, de nome MensagemPadrao, nela foi criada a classe TServicoMensagemPadrao, que implementa IServicoMensagem. Essa implementação padrão do serviço de mensagem utiliza a função Application.MessageBox, como mostra a **Listagem 10**.

Foi necessário realizar um mapeamento entre os tipos utilizados por IServicoMensagem e os tipos que Application.MessageBox espera. O primeiro está na linha 25, onde se adapta o tipo TBotoes para as constantes que representam os botões que irão surgir no diálogo. O segundo está na linha 31, determinando o tipo de mensagem. Isso afetará a imagem que é apresentada no diálogo, já o terceiro está na linha 41, onde a resposta padrão de Application.MessageBox é adaptada para o tipo TRetornoMensagem.

Listagem 10. Implementação padrão de IServicoMensagem

```
01 unit MensagemPadrao;
02
03 interface
04
05 uses IMensagem, Forms, Windows;
06 type
07  TServicoMensagemPadrao = class (TInterfacedObject, IServicoMensagem)
08  public
09    function Exibir(Mensagem: string;
10              TituloMensagem: string;
11              TipoMensagem: TTipoAlerta;
12              Botoes: TBotoes): TRetornoMensagem;
13  end;
14
15 implementation
16
17 { TServicoMensagemPadrao }
18
19 function TServicoMensagemPadrao.Exibir(Mensagem, TituloMensagem:
   string;
20  TipoMensagem: TTipoAlerta; Botoes: TBotoes): TRetornoMensagem;
21 var
22  opcoes: integer;
23  resposta: integer;
24 begin
25  case Botoes of
26   btnOk: opcoes := MB_OK ;
27   btnOkCancel: opcoes := MB_OKCANCEL ;
28   btnSimNao: opcoes := MB_YESNO ;
29  end;
30
31  case TipoMensagem of
32   taInformacao: opcoes := opcoes + MB_ICONINFORMATION;
33   taAlerta: opcoes := opcoes + MB_ICONWARNING ;
34   taErro: opcoes := opcoes + MB_ICONERROR ;
35  end;
36
37  resposta := Application.MessageBox(
38            PWideChar(Mensagem),
39            PWideChar(TituloMensagem),
40            opcoes);
41  case resposta of
42   ID_OK: result := rmOk;
43   ID_CANCEL: result := rmCancel;
44   ID_YES: result := rmSim;
45   ID_NO: result := rmNao;
46  end;
47 end;
```

Utilizando o serviço de mensagem

No formulário principal da aplicação temos um único botão que irá disparar o serviço de mensagem. Em seu evento click (**Listagem 11**) a interface IServicoMensagem é utilizada e não diretamente a classe TServicoPadrao.

Na linha 6 uma instância de TServicoMensagemPadrao é adicionada na variável servicoMensagem, isso só é possível porque a classe implementa a interface requerida. Caso seja necessário criar uma nova forma de exibição de mensagem, essa deve também implementar IServicoMensagem.

O código atual está bom, não dependemos de uma classe concreta, mas de uma interface. Isso por si só já garante flexibilidade. Porém existe um problema, a classe concreta é instanciada dentro do formulário, ou seja, ele conhece muito da implementação.

Mesmo se aqui fosse aplicado o conceito de injeção de dependência, a classe que injetaria essa dependência também conheceria muito da implementação. Como resolver isso então?

Listagem 11. Serviço de mensagem em uso

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 var
03   servicoMensagem: IServicoMensagem;
04   resultado: TRetornoMensagem;
05 begin
06   servicoMensagem := TServicoMensagemPadrao.Create;
07   resultado := servicoMensagem.Exibir('Teste de Mensagem, é legal?',
08                          'Responda',
09                          TTipoAlerta.taInformacao,
10                          TBotoes.btnSimNao);
11   case resultado of
12    rmSim: servicoMensagem.Exibir('Respondeu sim',
13                         'resposta',
14                         TTipoAlerta.taInformacao,
15                         TBotoes.btnOk);
16   rmNao: servicoMensagem.Exibir('Respondeu não',
17                         'resposta',
18                        TTipoAlerta.taInformacao,
19                         TBotoes.btnOk);
20  end;
21 end;
```

Implementando um container IoC

A solução é a utilização de um container IoC. Nele registramos as implementações (componentes) dos serviços utilizados (representados por interface). Como vamos utilizar o Spring4D, o primeiro passo é adicionar no Search Path do projeto o caminho do código fonte do Spring4D. Isso é feito ao acessar o menu Project>Options, como mostra a **Figura 3**. São cinco pastas necessárias: \Source\Base, \Source\Base\Collections, \Source\Base\Reflection, \Source\Core\Container, \Source\Core\Services. Também é necessário incluir uma única Unit no uses do projeto, a Spring.Container.

Então foi adicionado um segundo botão no formulário, em seu evento OnClick será instanciado um container, como mostra a **Listagem 12**.

Na linha 3 foi criada uma variável do tipo TContainer, que é o tipo que implementa todo o comportamento necessário de um container IoC, expondo quatro métodos básicos:

- **RegisterComponent** – Diz ao container, registra, qual componente implementou um determinado serviço e especifica o seu tipo de tempo de vida. No exemplo da linha 7 foi determinado que o componente instanciado em memória será um singleton (**BOX 3**), não sendo instanciado repetidamente a cada nova requisição;
- **Build** – Inspeciona todos os registros realizados e caso encontre uma referência circular, uma exceção é disparada;
- **Resolve** – O método Resolve<T> é utilizado para obter uma instância de um determinado serviço;
- **Release** – Quando uma instância não é mais utilizada, o método Release libera a instância do container.

Listagem 12. Utilizando um container IoC

```
01 procedure TForm1.Button2Click(Sender: TObject);
02 var
03   container: TContainer;
04   servicoMensagem: IServicoMensagem;
05 begin
06   container := TContainer.Create;
07   container.RegisterComponent<TServicoMensagemPadrao>
08     .Implements<IServicoMensagem>
09     .AsSingleton;
10   container.Build;
11
12   servicoMensagem := container.Resolve<IServicoMensagem>;
13   servicoMensagem.Exibir('Mensagem exibida via IoC',
14                'IoC Rocks!',
15                TTipoAlerta.taInformacao,
16                TBotoes.btnOk);
17   container.Release(servicoMensagem);
18   container.Free;
19 end;
```

BOX 3. Singleton

O Singleton é um padrão de projeto que tem a finalidade de resolver a seguinte questão: Como garantir que para uma determinada classe, haja apenas uma única instância dela.

Na linha 6 o container é instanciado e na linha 7 o registro do componente TServicoMensagemPadrao é feito, indicando que ele implementa o serviço IServicoMensagem. A linha 10 executa o método Build, que valida o registro realizado. Na linha 12 o container entra em execução. A chamar seu método Resolve passando a interface do serviço requerido, o container irá localizar qual componente está registrado para esse serviço, irá instanciá-lo e retorná-lo pronto para ser usado. A prova de que isso funciona está na linha 13, onde o serviço é executado. As linhas 17 e 18 apenas liberam a instância do serviço do container e o container em si.

Profissionalizando

O código pode ter parecido um pouco confuso, isso porque adicionamos mais responsabilidades do que deveríamos ao clique do botão, além de exibir a mensagem ele também está

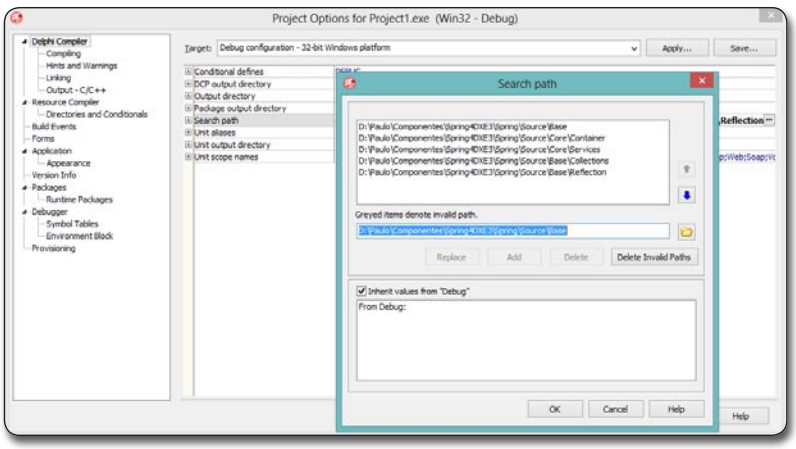

Figura 3. Search Path modificado

responsável por instanciar e configurar o container. Para organizar esse código movemos a variável container para a seção var do formulário, ou seja, ela se tornou uma variável pública. Removemos do evento Click o código de criação e configuração do container (linhas 6 a 10 da **Listagem 12**) e o movemos para a seção **initialization**. Essa seção é executada quando a Unit é carregada em execução, assim, o container estará disponível logo que o aplicativo é executado. Já o código de liberação de memória (linha 18) é movido para a seção **finalization**, que é executada quando a Unit é descarregada, garantindo sua liberação. Sendo assim, o que resta no evento Click, pode ser visto na **Listagem 13**.

Listagem 13. Evento Click otimizado

```
01 procedure TForm1.Button2Click(Sender: TObject);
02 var
03   servicoMensagem: IServicoMensagem;
04 begin
05   servicoMensagem := container.Resolve<IServicoMensagem>;
06   servicoMensagem.Exibir('Mensagem exibida via IoC',
07                 'IoC Rocks!',
08                 TTipoAlerta.taInformacao,
09                 TBotoes.btnOk);
10   container.Release(servicoMensagem);
11 end;
```

Agora o evento apenas realiza o acesso ao container, ele não sabe quem está implementando o serviço de mensagem.

Conclusão

Isolar o que varia através de uma interface garantiu a criação de várias formas de pagamento, já fazer uso do conceito de injeção de dependência permitiu centralizar as mudanças onde elas realmente precisam acontecer. Unir essas duas técnicas nos permite ir além, esse tipo de implementação nos permite inclusive, isolar as classes de Forma de Pagamento em uma DLL, para serem utilizadas em outras aplicações, se assim for o caso.

Utilizar um container para centralizar a inversão de controle é uma prática largamente utilizada por desenvolvedores de outras plataformas como Java e .NET, sendo considerada uma técnica mais que recomendada para garantir flexibilidade. Talvez essa questão de flexibilidade não tenha ficado clara, mas imaginemos um sistema que realiza cálculos de acordo com a unidade federativa em uso. Se o cliente é de São Paulo o cálculo é de uma maneira, se é do Rio de Janeiro é de outra. Esse cálculo poderia ser abstraído como um serviço (através de uma Interface), e duas classes concretas desse serviço seriam criadas. Uma para cada unidade federativa. Essas são os componentes. Agora, durante a execução, no momento de registrar quem implementa o serviço, bastaria identificar a unidade federativa em uso e registrar o componente adequado. Todo o código do sistema que faz uso do serviço de cálculo não sabe se a implementação é São Paulo ou do Rio de Janeiro, quem sabe isso é o container, assim, é necessário apenas requisitá-lo uma instância e essa será retornar de acordo com a regra de negócio envolvida, nenhuma outra parte do sistema precisa ser alterada. Se esse cálculo precisar ser modificado para outra unidade federativa, basta criar esse novo componente e incluí-lo na configuração do container. Isso é flexibilidade.

Autor

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas da Siplan Control-M (www.siplancontrolm.com.br) unidade Jaboticabal, e professor da FATEC, campus Taquaritinga. Blog: http://quicoli.worpress.com

Links:

Delphi-Mock
https://github.com/VSoftTechnologies/Delphi-Mocks

Spring4D
http://www.spring4d.org/

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

Feita para Desenvolvedores de Software e DBAs
SQL magazine

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

http://www.devmedia.com.br/cursos/banco-de-dados

http://www.devmedia.com.br

DEVMEDIA

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486